UNIVERSITÉ DE MONTRÉAL

ANALYSE DE L'EXÉCUTION DE SYSTÈMES VIRTUALISÉS MULTINIVEAUX

CÉDRIC BIANCHERI
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)
AOÛT 2016

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ANALYSE DE L'EXÉCUTION DE SYSTÈMES VIRTUALISÉS MULTINIVEAUX

présenté par : BIANCHERI Cédric
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

REMERCIEMENTS

J'aimerais remercier toutes les personnes reliées de près ou de loin au DORSAL qui m'ont permis de réaliser ce projet. Leur soutien et conseils m'ont été précieux.

Je remercie aussi et surtout ma famille, pour m'avoir encouragé tout au long de mes études, malgré la distance et le temps bien trop court passé avec elle. Elle est une source d'inspiration et de motivation. Sans elle, je n'aurais jamais pu arriver là où je suis aujourd'hui.

RÉSUMÉ

Au cours des dernières années, l'utilisation de systèmes virtualisés s'est fortement développée, que ça soit dans l'industrie, ou par l'usage de particuliers. Cette globalisation est notamment due au développement des technologies reliées, améliorant toujours plus les performances de tels systèmes. La flexibilité des systèmes virtualisés s'ajoute également aux raisons de cet essor. Cette technologie permet de facilement et rapidement gérer tout un parc de machines virtuelles (VMs) et conteneurs, répartis sur les nœuds d'un réseau de machines physiques. La gestion des ressources a un impact moins onéreux pour son utilisateur car elle permet une adaptation à la charge de travail demandée. Même si les machines virtuelles et les conteneurs offrent une certaine isolation de leur système hôte, toute activité survient éventuellement sur un processeur (CPU) physique d'un des nœuds. Ainsi, VMs et conteneurs partagent un même ensemble de ressources, sans pour autant en avoir conscience. Ce partage peut entraîner un surengagement des ressources, impliquant pour un ou plusieurs systèmes, une baisse de performance. Pour avoir une meilleure approche de l'exécution réelle sur un nœud de réseau, notre étude se donne pour objectif de fournir une représentation, aussi fidèle qu'elle soit, du flot d'exécution survenu sur ses processeurs, malgré les différentes couches de virtualisation.

Pour atteindre cet objectif, nous nous servons du traceur LTTng afin de générer et exploiter les traces noyau des systèmes hôte et invités. Les traces noyau fournissent de nombreuses informations, à très bas niveau, sur l'activité d'un système d'exploitation. Elles nous permettent, entre autres, de déterminer les phases où un système virtualisé s'approprie une ressource telle qu'un processeur physique. En plus du flot réel d'exécution, l'exploitation de telles données nous permet de recréer la hiérarchie complète de conteneurs et jusqu'à deux niveaux de machines virtuelles imbriquées.

Pour compléter cette étude, nous présentons une vue, développée dans le logiciel Trace Compass, capable de représenter l'ensemble des informations extraites par la précédente analyse. Cette vue permet de visualiser l'ensemble de la hiérarchie des systèmes tracés, de même que leurs flots d'exécution sur les CPUs physique du nœud ciblé. Finalement, par l'utilisation de filtres, il est également possible de mettre en relief des aspects plus précis des systèmes tracés, comme une VM, un CPU virtuel, un conteneur ou un ensemble de processus.

ABSTRACT

Over the past few years, the use of virtualized systems has developed significantly, in industry as well as for personal usage. This global deployment is mainly due to the development of the related technologies, allowing further improvement in the performance of such systems. The flexibility of virtualized systems also contributes to this growth. This technology allows the easy and quick management of large clusters of virtual machines (VMs) and containers, spread over the nodes of a physical machine network. Furthermore, by adapting to changing workloads, the management of physical resources reduces hardware and energy consumption costs. Even if VMs and containers are isolated from their host, all their activity eventually happens on a node's physical CPU. Thus, VMs and containers might unknowingly share the same physical resources, resulting in an overcommitment of those resources and a performance drop. For a better oversight of the real execution on one of the network's node, our study aims to represent an accurate flow of the activities that occurred on its processor cores, erasing the bounds of the virtualization layers.

To fulfill this objective, we use the LTTng tracer to generate kernel traces on the host and the guest systems. Kernel traces can provide plenty of fine-grained information, regarding the activity of the operating system. Among others, they allow us to determine the stages when a virtualized system yields a resource such as a physical CPU. In addition to recovering the real execution flow, the analysis of this information allows to reestablish the full hierarchy of containers, and up to two layers of nested VMs.

To complete this study, we present a view developed in Trace Compass, able to represent the full extent of the information extracted by the analysis. This view allows to visualize the hierarchy of the traced systems, as well as their execution flow on the physical CPUs of their corresponding node. Finally, by using a system of filters, we offer the possibility to highlight more details of the traced systems, such as the execution flow of a VM, a virtual CPU, a container, or a subset of processes.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

AMD	Advanced Micro Devices (marque de commerce)
CPU	Central Processing Unit
CTF	Common Trace Format
ETW	Event Tracing for Windows
FVS	Fused Virtualized Systems
IaaS	Infrastructure as a Service
IBM	International Business Machines Corporation
IO	Input/Output
JVM	Java Virtual Machine
KVM	Kernel-based Virtual Machine
LTTng	Linux Trace Toolkit - next generation
LXC	Linux Containers
NSID	NameSpace Identifier
NVM	Nested VM
PCPU	Physical CPU
PID	Process Identifier
SLVM	Single Layered VM
SHT	State History Tree
SV	Système Virtualisé
TCP	Transmission Control Protocol
TID	Thread Identifier
UST	UserSpace Tracing
vCPU	vitual CPU
VMCS	Virtual Machine Control Structure
VM	Virtual Machine
vTID	virtual TID
VS	Virtualized System

CHAPITRE 1

INTRODUCTION

Le développement de l'infonuagique (Cloud Computing), au cours des dernières années, est dû à la possibilité de créer des modèles flexibles, des environnements de test, et surtout à son caractère moins onéreux vis-à-vis de la maintenance et de la consommation d'énergie. Cependant, cette technologie apporte avec elle son lot de défis relatifs à la détection et le diagnostic d'anomalies d'exécution. Dans le cas d'une architecture n'utilisant pas de niveaux de virtualisation, une connaissance minimale de l'activité de la machine physique permet d'effectuer une analyse concluante. Par exemple, si l'on sait, à tout instant, quel processus s'exécute sur un CPU, il est possible de déterminer lesquels ont ralenti l'exécution d'une tâche ciblée. Cependant, il devient plus compliqué d'arriver au même résultat avec des architectures multiniveaux. En effet, l'utilisation de systèmes virtualisés (SVs) engendre des flots d'exécution indépendant les uns des autres. Certaines tâches peuvent alors être interrompues, sans que le système les ayant initiées ne s'en rende compte.

Ce travail se concentre sur l'étude de traces noyaux provenant d'une machine physique, ses machines virtuelles et ses conteneurs Linux (LXC), pour permettre de fournir un outil capable d'identifier des anomalies d'exécution. L'idée est de permettre à un utilisateur d'observer toute la hiérarchie de ses SVs, comme si l'intégralité des flots d'exécution se déroulait sur les CPUs physiques du système hôte. L'intérêt est de simplifier l'observation de systèmes complexes à l'aide de traces de très bas niveau. Avant d'entrer dans les détails de ce travail, nous allons énoncer quelques notions clés facilitant la bonne compréhension du sujet.

1.1 Concepts de base

1.1.1 Machine virtuelle

Une machine virtuelle (ou VM) est souvent associée à un contexte d'exécution isolé des autres flots. Cette isolation sert de couche d'abstraction offrant une plus grande flexibilité d'exécution pour la machine. Ainsi, des technologies comme la machine virtuelle Java, les conteneurs, ou les systèmes d'exploitation virtualisés, peuvent toutes être désignées comme des machines virtuelles. Cependant, en infonuagique et dans le cadre de ce travail, nous nous limiterons au cas de systèmes d'exploitation virtualisés, hébergés par un hôte pouvant être aussi bien une machine physique qu'une VM à son tour. L'abstraction entre la VM et son

hôte permet une dissociation entre ces derniers. De cette manière, il devient possible de figer, réveiller ou même migrer une VM vers un nouvel hôte, sans avoir besoin de l'arrêter. La virtualisation permet aussi d'associer précisément des ressources à une VM telles que des processeurs, de la mémoire vive, de l'espace disque ou de la bande passante. Cette technologie a permis le développement des plateformes en tant que service (PaaS), où des fournisseurs de services infonuagiques mettent des machines virtuelles à disposition de clients qui n'auront pas à se préoccuper du partage de ressources physiques avec d'autres machines.

1.1.2 Conteneur

Un conteneur est aussi un système virtualisé. La principale différence avec une VM réside dans le fait que le conteneur partage son noyau avec son hôte, contrairement à la VM qui possède son noyau propre. Le système est isolé par ce que l'on appelle des espaces de nommage. Par exemple, les processus d'un conteneur ont un identifiant unique dans leur espace de nommage et ne peuvent pas accéder aux processus appartenant à un espace de nommage autre que le leur ou contenu dans le leur. Par conséquent, un processus aura un identifiant pour chaque conteneur auquel il appartient. Le principal avantage des conteneurs est que l'exécution se fait nativement sur l'hôte. Il n'y a pas réellement de virtualisation comme pour les machines virtuelles. Cela rend les conteneurs beaucoup plus performants que ces dernières qui souffrent du surcoût de la virtualisation.

1.1.3 Évènement

Un évènement est un instant de l'espace-temps. On lui associe une estampille de temps, qui permet de savoir quand il a été émis, et une ressource, qui identifie la source de l'émission. Un identifiant lui est associé, pour donner un sens à son existence, ainsi que parfois une charge utile, contenant des informations propres à l'évènement en question. Un évènement peut servir à donner une indication sur un état ponctuel ou prolongé du système. Dans le second cas, l'état sera alors délimité par des évènements de début et de fin. Dans notre cas, les évènements seront générés par les processus du système et seront enregistrés pour former une trace.

1.1.4 Traçage

Le traçage est l'action d'enregistrer des évènements dans le but de former une trace. Cette trace sera par la suite analysée dans l'optique de mieux comprendre le système tracé. Pour tracer une application, il faut placer dans celle-ci des points qui généreront un évènement à

chaque fois qu'ils seront rencontrés. Dans le même ordre d'idée que le traçage d'application, il est aussi possible de tracer le noyau d'un système d'exploitation. Le traçage noyau permet de connaître le comportement à très bas niveau d'une application sans avoir besoin de modifier son code. Dans le cadre de cette étude, nous utiliserons exclusivement des traces noyau pour rester indépendant de l'instrumentation des applications en espace utilisateur.

1.1.5 Attribut

Un attribut est une composante du modèle généré suite à l'analyse d'une trace. Dans cette étude, les attributs feront partie d'un arbre d'attributs, aussi appelé *state system*, qui modélisera l'état du système pour toute la durée de la trace. Chaque nœud de l'arbre est un attribut qui possède obligatoirement un nom, et une valeur facultative. Bien que le nom soit fixe à tout instant, la valeur peut changer au cours du temps. Pour récupérer la valeur d'un attribut, il faut fournir l'intégralité des noms des attributs parents, ainsi qu'une estampille de temps. Par exemple, pour connaître la valeur de l'attribut `"State"`, du CPU 0 de la machine Host à l'instant t, il faudra utiliser le chemin `"Host/CPUs/CPU0/State"` avec l'estampille t.

1.2 Problème étudié et buts poursuivis

1.2.1 Éléments de la problématique

L'avantage de l'isolation des machines virtuelles s'avère devenir une contrainte quand elle est confrontée au besoin d'analyser des systèmes virtualisés. La couche de virtualisation empêche un hôte de connaître le flot d'exécution de ses machines virtuelles. Inversement, une VM peut ne pas avoir conscience de l'existence de son hôte et des autres machines avec lesquelles elle partage les ressources physiques. Ainsi, analyser des traces de l'hôte et ses machines virtuelles séparément donnera des résultats incomplets, biaisés par la couche de virtualisation. Par exemple, une VM partageant un processeur physique avec une autre VM, risque de voir son temps CPU volé par la seconde machine, sans que cela n'apparaisse dans sa trace. De son côté, la VM verra que son CPU virtuel est utilisé alors qu'il est en réalité en attente. Un temps d'exécution anormalement long pour la VM sera le seul symptôme de cette préemption.

Pour les conteneurs lancés directement sur l'hôte, la difficulté est moindre car tracer le noyau de l'hôte revient à également tracer le conteneur. Il faut cependant être capable d'établir l'existence du conteneur et de reconnaître quelles sont les tâches qui lui sont associées. La problématique énoncée pour les VMs reste malgré tout présente dans le cas des conteneurs lancés à partir d'une VM.

La question que l'on peut alors se poser est de savoir s'il est possible, à partir de traces noyau, de retrouver l'intégralité de la hiérarchie des systèmes virtualisés étudiés, ainsi que le flot réel d'exécution sur les processeurs physiques de l'hôte. Car même s'il se trouve à l'intérieur d'un conteneur, une VM lancée depuis l'hôte, ou dans des VMs imbriquées, un processus ira toujours s'exécuter sur un CPU de l'hôte.

1.2.2 Objectifs de recherche

Pour résumer, nous avons un principal objectif qui est de briser les frontières entre les différents systèmes virtualisés pour ramener tous les différents niveaux à celui de l'hôte. Cet objectif est subdivisé en les sous-objectifs suivant :

1. Envisager les différentes stratégies possibles pour analyser les traces noyau des différentes machines ;
2. Déterminer quand une VM s'exécute sur un CPU physique ;
3. Proposer un arbre d'attributs contenant à la fois des informations sur l'hôte et ses VMs ;
4. Créer une vue capable de présenter clairement les informations contenues dans l'arbre d'attributs ;
5. Généraliser l'analyse aux VMs imbriquées et aux conteneurs ;

1.3 Démarche de l'ensemble du travail

Comme le laissent sous-entendre les objectifs de recherche, le travail présenté est divisé en deux parties complémentaires. La première partie consiste à analyser les traces effectuées comme un tout, et non pas individuellement, pour pouvoir retrouver le flot d'exécution réel de l'ensemble des systèmes virtualisés. Cela permet alors de créer un modèle synthétisant l'intégralité des niveaux de virtualisation pour la durée de la trace.

La seconde partie a pour but d'exploiter le modèle créé en lui associant une vue adaptée à ses spécificités. Cette vue doit permettre l'accès aux informations contenues dans le modèle tout en gardant une bonne lisibilité et fluidité d'utilisation.

1.4 Plan du mémoire

Le chapite 2 présente une revue de littérature des thèmes abordés dans ces travaux. Le chapitre 3 apporte plus de clarification quant à la démarche employée dans la progression

des travaux présentés. Le chapitre 4 intègre le contenu de l'article «Fine-grained Multilayer Virtualized Systems Analysis» qui reprend les points des objectifs de recherche. Le chapitre 5 permet de revenir sur des points de l'article. Finalement, le chapitre 6 fait une synthèse des travaux et clôture sur des pistes envisageables pour des travaux futurs.

CHAPITRE 2

REVUE DE LITTÉRATURE

2.1 Traçage

Le but du traçage est de rassembler un maximum d'information sur un système tout en minimisant son impact sur celui-ci. Contrairement à des méthodes qui s'intéressent à la différence entre deux états du système ou à un instant donné, le traçage sert à comprendre le fonctionnement d'un système durant son exécution. Il doit donc être le moins intrusif possible pour que la trace créée soit une représentation fidèle de l'activité réelle du système.

2.1.1 Instrumentation

Le traçage est réalisé par l'instrumentation du code que l'on souhaite tracer. Cette instrumentation peut être réalisée de deux façons.

Instrumentation dynamique

L'instrumentation dynamique est une méthode d'instrumentation qui ne nécessite pas de modifier le code de l'application. De ce fait, il devient alors inutile de recompiler le programme ciblé pour changer le comportement des points de trace. kprobes [1] réalise cela dans l'espace noyau par la mise en place d'un mécanisme injectant, à l'endroit souhaité, un point d'arrêt avec le code le traitant. De cette façon, il est alors possible d'instrumenter dans le noyau n'importe quelle instruction, ainsi que les entrées et les sorties de fonctions. Ce mécanisme a aussi été étendu à l'espace utilisateur grâce à uprobes [2]. Bien que très pratique car ne nécessitant pas de recompiler le code source, cette méthode d'instrumentation a un coût plus élevé que l'instrumentation statique.

Instrumentation statique

À l'inverse de l'instrumentation dynamique, l'instrumentation statique se base sur l'insertion de points de trace directement dans le fichier binaire d'une application. Pour utiliser ce type d'instrumentation il faudra alors nécessairement recompiler le code instrumenté. Le noyau Linux est instrumenté statiquement grâce à la macro `TRACE_EVENT` [3]. Celle-ci crée des points de trace capables d'enregistrer des informations relatives au contexte dans lequel

ils ont été atteints pour pouvoir, par la suite, les transmettre au traceur. Par ailleurs, cette macro a été développée pour être indépendante du traceur utilisé et est donc utilisée par grand nombre d'entre eux parmi lesquels on peut citer perf, LTTng, Ftrace et SystemTap.

2.1.2 Traceurs Linux

Le rôle du traceur est d'exploiter les points de trace placés dans le code instrumenté. Il doit détecter quand ils sont atteints et enregistrer leur occurrence pour générer la trace. Il va de soit que cette étape doit s'effectuer de la façon la moins invasive possible pour ne pas perturber l'exécution du système, ce qui pourrait compromettre les données collectées. Par exemple, une erreur se produisant suite au lancement quasi-simultané de deux tâches, pourrait ne pas être observée lors du traçage, si le traceur dégrade les performances du système.

LTTng

Le traceur LTTng [4] est un ensemble de trois composantes, capable de tracer aussi bien en espace noyau, grâce à son instrumentation statique, qu'en espace utilisateur, avec un faible surcoût.

Sa première composante, LTTng-tools, est le cœur du traceur car elle contient les outils nécessaires pour orchestrer le traçage. On y trouve un *session daemon*, responsable de la gestion des sessions de traçage, qui a parmi ses principales tâches, celle de contrôler les évènements activés. Le *consumer daemon* est chargé de collecter les évènements enregistrés pour les écrire sur disque ou bien les envoyer par le réseau par l'intermédiaire d'un *relay daemon*. La trace générée est au format CTF, un format binaire ayant pour but de devenir un standard pour les logiciels de traçage. La collecte des évènements se fait via des tampons circulaires, eux-mêmes subdivisés en sous-tampons. Cette architecture permet une flexibilité dans les paramètres de traçage car elle autorise l'utilisateur à modifier la taille et le nombre des sous-tampons pour s'adapter au système tracé. Par ailleurs, l'utilisateur a aussi le choix entre deux stratégies d'utilisation du tampon circulaire. Il peut d'une part choisir de jeter les nouveaux évènements quand le tampon est plein, soit d'écraser ceux qui n'ont pas pu encore être traités. Ces deux stratégies non bloquantes permettent de minimiser l'impact du traceur.

La seconde composante, LTTng-modules, est un ensemble de modules noyau servant à tracer le noyau Linux. C'est eux qui définissent les fonctions de rappels, appelées quand un point de trace est atteint, ainsi que le tampon circulaire.

LTTng-UST est la dernière composante du traceur LTTng. Elle est dédiée au traçage en espace utilisateur. Un utilisateur peu simplement créer un point de trace comme s'il s'agissait d'un *printf* avec l'aide de *tracef*. Mais s'il veut plus de flexibilité et de contrôle sur différents paramètres, comme le nom de ses points trace ou le type des éléments dans la charge utile, il devra éditer manuellement des fichiers ou bien utiliser l'outil *lttng-gen-tp* qui permet de faire cela plus simplement. Comme on est dans un cas d'instrumentation statique, il faudra recompiler l'application après l'ajout des points de trace. Il est possible d'installer LTTng-UST sans LTTng-modules et vice versa. En effet le traceur peut être utilisé uniquement pour le noyau Linux ou l'espace utilisateur.

Les traces générées par LTTng peuvent être lues par l'utilitaire babeltrace. Le résultat en sortie est de la forme suivante :

```
[14:46:37.973190412] (+0.000000088) server1 sched_switch: { cpu_id = 0 },
{ prev_comm = "lttng-consumerd", prev_tid = 1707, prev_prio = 20,
    prev_state = 2, next_comm = "vm_forks", next_tid = 2548,
    next_prio = 20 }
[14:46:37.973197973] (+0.000000238) server1 sched_stat_blocked: { cpu_id = 0 },
    { comm = "lttng-consumerd", tid = 1707, delay = 9274 }
[14:46:37.973198817] (+0.000000744) server1 sched_wakeup: { cpu_id = 0 },
    { comm = "lttng-consumerd", tid = 1707, prio = 120, success = 1,
    target_cpu = 0 }
[14:46:37.973199425] (+0.000000608) server1 rcu_utilization: { cpu_id = 0 },
    { s = "Start context switch" }
[14:46:37.973199587] (+0.000000162) server1 rcu_utilization: { cpu_id = 0 },
    { s = "End context switch" }
[14:46:37.973199986] (+0.000000399) server1 sched_stat_runtime: { cpu_id = 0 },
    { comm = "vm_forks", tid = 2548, runtime = 9274,
    vruntime = 1907572826 }
[14:46:37.973200395] (+0.000000409) server1 sched_stat_wait: { cpu_id = 0 },
    { comm = "lttng-consumerd", tid = 1707, delay = 0 }
[14:46:37.973200829] (+0.000000434) server1 sched_switch: { cpu_id = 0 },
    { prev_comm = "vm_forks", prev_tid = 2548, prev_prio = 20,
    prev_state = 0, next_comm = "lttng-consumerd", next_tid = 1707,
    next_prio = 20 }
[14:46:37.973202386] (+0.000001557) server1 syscall_exit_fadvise64:
    { cpu_id = 0 }, { ret = 0 }
```

On peut voir, dans cet extrait, le *consumer daemon* rendre le CPU 0 au processus *vm_forks* puis se faire réveiller pour finalement revenir prendre le CPU et terminer l'appel système *fadvise64*.

C'est ce traceur qui sera utilisé dans nos travaux.

Ftrace

Contrairement à LTTng, Ftrace [5] est un traceur entièrement inclus dans le noyau Linux et précédemment dans le noyau Linux temps réel. Il n'est donc pas nécessaire d'installer des outils pour pouvoir l'utiliser. Il utilise lui aussi la macro `TRACE_EVENT` et a la particularité de pouvoir ajouter des points de trace dynamiquement. Créé à la base pour tracer les appels de fonctions du noyau pour extraire le chemin critique sur un intervalle de temps, il a évolué pour permettre d'autres fonctionnalités comme l'analyse de latence, des désactivations et réactivations des interruptions, ou de l'ordonnanceur.

Son principe de traçage de fonctions est lié à l'appel d'une fonction mcount(), injecté dans le code de chaque fonction du noyau Linux. Lors de la compilation du noyau avec l'option `CONFIG_DYNAMIC_FTRACE`, chaque appel la fonction mcount() est remplacé par l'instruction `NOP`. L'instruction n'ayant aucun effet si ce n'est d'incrémenter le pointeur d'instruction, cela permet de n'avoir aucune perte de performance quand le noyau n'est pas tracé. C'est durant la phase de compilation que chaque adresse où est injecté l'appel à mcount() est enregistré dans une liste. Cette liste permet d'une part de faire au démarrage le remplacement précédemment mentionné, et d'autre part de réaliser l'interversion opposée quand on souhaite activer le traçage des fonctions. Suivant ce procédé, Ftrace peut finement filtrer les fonctions à tracer.

Ftrace s'utilise via le système de fichiers *debugfs*, accessible par l'intermédiaire du répertoire `/sys/kernel/debug`. Toute la configuration et la manipulation du traçage se fait alors en écrivant dans les différents fichiers du dossier `tracing`. Par exemple, l'affichage des différents types de traçage et la sélection de l'un d'eux, se font avec les commandes suivantes :

```
# cat  available_tracers
blk mmiotrace function_graph wakeup_dl wakeup_rt wakeup function nop
# echo function > current_tracer
```

L'activation du traçage se fait par l'écriture d'un 1 dans le fichier `tracing_enabled`. Suite à cela, la trace peut être visualisée dans trois fichiers :

— `trace`, pour la lecture directe de la trace,
— `latency_trace`, pour mettre en avant des informations liées à la détection de problèmes de latence,
— `trace_pipe`, également pour la lecture de la trace mais par l'intermédiaire d'un pipe.

Ci-dessous, on peut observer un exemple d'affichage des vingt premières lignes d'une trace effectuée avec le traceur `function`.

```
# cat trace | head -20
# tracer: function
#
# entries-in-buffer/entries-written: 1295/1295   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#       TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#          | |       |   ||||       |         |
        bash-14408 [000] .... 34745.040668: do_dup2 <-SyS_dup2
        bash-14408 [000] .... 34745.040669: filp_close <-do_dup2
        bash-14408 [000] .... 34745.040669: dnotify_flush <-filp_close
        bash-14408 [000] .... 34745.040669: locks_remove_posix <-filp_close
        bash-14408 [000] .... 34745.040669: fput <-filp_close
        bash-14408 [000] .... 34745.040670: task_work_add <-fput
        bash-14408 [000] .... 34745.040670: kick_process <-task_work_add
        bash-14408 [000] .... 34745.040670: do_notify_resume <-int_signal
        bash-14408 [000] .... 34745.040670: task_work_run <-do_notify_resume
```

L'affichage permet d'observer facilement l'appel de chaque fonction de même que la fonction appelante, le tout en affichant également le processus responsable de l'appel ainsi que l'estampille de temps associée.

Le traceur `function_graph` offre un autre aperçu du traçage de fonctions que `function` :

```
# echo function_graph > current_tracer
# echo 1 > tracing_on ; echo "Hello World!"; echo 0 > tracing_on
# cat trace | head -16
```

```
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 1)               |  menu_select() {
 0)   0.193 us    |  mutex_unlock();
 1)   0.064 us    |    pm_qos_request();
 0)   0.048 us    |  __fsnotify_parent();
 1)   0.054 us    |    tick_nohz_get_sleep_length();
 0)               |  fsnotify() {
 0)   0.053 us    |    __srcu_read_lock();
 1)   0.038 us    |    get_iowait_load();
 0)   0.042 us    |    __srcu_read_unlock();
 0)   0.636 us    |  }
 0)   0.053 us    |  __sb_end_write();
 1)   2.065 us    |  }
```

La hiérarchie d'appels est présentée de façon arborescente, montrant le temps passé dans chaque fonction. Ce temps inclut le temps passé dans une fonction, les fonctions qu'elle a appelées et le surcoût dû au traçage.

Même si l'utilisation par le système de fichiers *debugfs* est facile, il faut reconnaître qu'elle n'est pas forcément très pratique. trace-cmd [6] est un utilitaire spécialement conçu pour utiliser Ftrace. Par exemple, pour réaliser le même tracé que précédemment, il suffit d'utiliser la commande suivante :

```
# trace-cmd record -p function_graph -o ~/trace.dat echo "Hello World!"
```

Le résultat sera écrit au format binaire dans le fichier `trace.dat`, lisible soit par la commande report de trace-cmd, soit par un utilitaire graphique comme KernelShark [7]. À son lancement, trace-cmd crée un processus par processeur. Comme les tampons circulaires de chaque CPU utilisés par Ftrace sont directement mappés avec les fichiers `trace_pipe_raw`, chacun des processus créés ira alors ouvrir le fichier `trace_pipe_raw` de son processeur associé pour enregistrer les évènements dans un fichier intermédiaire. Ce n'est qu'à la fin du traçage que les différents fichiers seront fusionnés.

Perf

Comme son nom l'indique, Perf [8] est à l'origine un outil d'analyse de performance. Tout comme Ftrace, il est intégré au noyau Linux. Il est principalement utilisé pour donner des statistiques sur l'exécution du système comme par exemple le nombre de fautes de page, de migrations de CPU, de changements de contexte ou bien aussi la liste des fonctions les plus utilisées. Ces statistiques proviennent de compteurs de performance matériels et logiciels. Ci-dessous on peut observer le résultat de base d'une utilisation de perf pendant l'affichage dans un terminal d'un document.

```
$perf stat cat document.txt
Performance counter stats for 'cat document.txt':
          0,299142  task-clock (msec)       0,572 CPUs utilized
                 5  context-switches        0,017 M/sec
                 0  cpu-migrations          0,000 K/sec
                69  page-faults             0,231 M/sec
           964 759  cycles                  3,225 GHz
           729 168  instructions            0,76  insns per cycle
           150 040  branches                501,568 M/sec
             7 316  branch-misses           4,88% of all branches

       0,000523169 seconds time elapsed
```

SystemTap

À l'instar de LTTng, SystemTap est capable de tracer à la fois en espace noyau et en espace utilisateur. Dans les deux cas c'est grâce à l'utilisation de kprobes ou uprobes, c'est à dire de l'instrumentation dynamique, qu'il y parvient. Il a été principalement conçu pour faciliter l'instrumentation du noyau qui nécessite une certaine expertise. Cependant, permettre une grande flexibilité peut avoir un coût. SystemTap s'adapte mal aux applications multi-thread, comme le souligne l'étude faite par Desnoyer et Desfossez [9]. SystemTap s'utilise par l'intermédiaire d'un langage de script, ressemblant aux langages awk et C. Conçu pour être léger, il rend l'utilisation des point-virgules, ainsi que la déclaration des types de données, optionnelle. Il est axé autour de deux points, les sondes et leurs effets. La sonde sert à définir un point dans le noyau, tandis que l'effet est l'action devant être exécutée lorsque le point

est atteint.

Uscope

Uscope [10] est un traceur avec pour but d'unifier le traçage noyau avec le traçage en espace utilisateur. Les traceurs, comme LTTng ou SystemTap, nécessitent la prédétermination du code fautif d'une application pour pouvoir ensuite l'instrumenter. Or, il n'est pas toujours facile d'arriver à déterminer à l'avance quelle partie du code est problématique car cela nécessite de nombreuses itérations avant de trouver où placer les points de trace. C'est une approche dite de l'espace utilisateur vers l'espace noyau. Uscope propose d'inverser cette approche en allant cette fois du noyau vers l'espace utilisateur en utilisant une version optimisée de *stack walking*. Cette approche est schématisée dans la figure 2.1. De cette façon, Uscope arrive à collecter, avec seulement 6% de surcoût, des informations en espace utilisateur à chaque fois qu'un point de trace du noyau est déclenché. Cette approche est schématisée dans la figure 2.1. Cette méthode est pratique pour l'analyse de boîtes noires qui ne peut reposer que sur le traçage noyau.

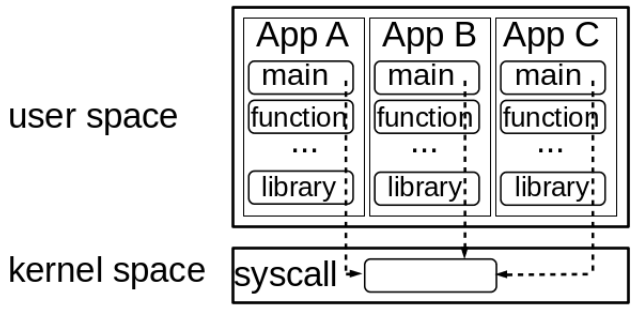

Figure 2.1 Approche noyau vers espace utilisateur employée par Uscope.

2.1.3 Traçage sous Windows

Windows permet le traçage noyau et en espace utilisateur par l'intermédiaire de la plateforme Event Tracing for Windows (ETW) [11]. Son fonctionnement est assez similaire à LTTng, notamment par la présence d'un système de tampons stockant les évènements avant de les envoyer à un consommateur, qui affichera les données en temps réel ou les enregistrera dans un fichier de trace. Comme LTTng avec son *state dump*, il génère de l'information au début de chaque trace représentant l'état actuel du système. De plus, par l'intermédiaire d'une

interface de programmation, il permet à n'importe quelle application de devenir producteur d'évènements.

2.1.4 Analyse de trace

Pour obtenir des informations spécifiques liées à une trace, il faut mettre en place une analyse. Les traces sont souvent très volumineuses. Par exemple, une trace noyau peut facilement contenir plusieurs centaines de milliers d'évènements, seulement pour une durée de traçage de quelques secondes. C'est donc à la fois un défi de tracer un système sans trop le perturber, mais aussi d'analyser la trace créée dans un temps raisonnable, sans monopoliser l'intégralité des ressources.

Les visualiseurs de traces nécessitent la connaissance de l'état du système en tout temps. Cependant, il n'est pas concevable de calculer et stocker l'ensemble des états du système avec une granularité aussi fine que celle des évènements tracés. En effet, cela reviendrait à stocker des millions d'états par seconde, monopolisant la mémoire du système ainsi que le temps de l'utilisateur. Une solution possible est alors de ne calculer l'état du système, ou cliché, qu'à intervalles réguliers. Pour retrouver les états intermédiaires, par exemple quand on zoome dans une vue, il suffit alors de sélectionner l'état du système sauvegardé le plus proche et de rejouer l'analyse de la trace à partir de ce point. Pour garder un temps de requête constant, chaque cliché peut ne pas être séparé par une durée de temps égale, mais par une même quantité d'évènements. Cette méthode est suffisante pour des traces de petites tailles. Cependant, avec des traces de grande taille, le nombre et la taille des clichés peuvent augmenter drastiquement, générant un problème d'utilisation de la mémoire.

C'est suite à ce constat que Montplaisir et al. [12] introduisent l'arbre à historique. Cette structure permet d'assurer un accès aux états d'un système en $\mathcal{O}(\log n)$, sans stocker d'information redondante. Pour réaliser cela, chaque état du système est représenté par un intervalle, symbolisant ses temps de début et de fin, pour être stocké sur disque. Les intervalles étant triés par ordre croissant par rapport à leur temps de fin, la construction de l'arbre se fait de façon incrémentale et permet donc d'être arrêtée et reprise à tout moment. De plus, cette structure a l'avantage de ne pas avoir besoin d'être rééquilibrée lors de sa construction, réduisant ainsi le temps nécessaire pour mener à bout l'analyse. Dans leurs travaux, Montplaisir et al. ont fait une étude comparative avec d'autres structures arborescentes et ont mis en évidence que certaines structures, comme les R-tree [13], ne peuvent réaliser l'analyse d'une trace de plusieurs centaines de mégaoctets dans un temps raisonnable à cause de ce rééquilibrage. Une autre conséquence est la possibilité d'envoyer des requêtes à l'arbre avant même qu'il soit complet. Cela permet ainsi à une vue de commencer à l'utiliser sans attendre la fin

de sa construction.

C'est cette structure qui est actuellement utilisée dans Trace Compass et que nous utiliserons dans nos travaux.

2.2 Virtualisation

Le partage de ressources est une notion récurrente en informatique. Son but est de permettre l'optimisation du système en distribuant les ressources physiques entre différents acteurs, tout en minimisant leur temps d'attente et la sous-utilisation des ressources disponibles. Bien que cela soit partiellement résolu grâce à l'introduction de l'ordonnancement, la virtualisation se permet d'aller plus loin dans le partage de ressources physiques. Elle permet une dissociation avec la machine physique impliquant une gestion plus flexible du système virtualisé. Il n'est plus nécessaire de consacrer une machine physique pour un seul service. Des systèmes virtualisés peuvent cohabiter sans avoir conscience qu'ils ne sont pas les seuls à s'exécuter. Les gestionnaires de parcs informatiques peuvent alors réduire leurs coûts de maintenance, car il y a moins de machines physiques, et également leur consommation d'énergie en adaptant le nombre de ressources disponibles en fonction de la demande.

La flexibilité des systèmes virtualisés ne se trouve pas seulement dans la façon de distribuer les ressources mais aussi dans la façon de les exploiter. Grâce à l'émulation, il devient possible d'exécuter du code prévu pour une certaine plateforme sans posséder physiquement celle-ci. Par exemple, le système d'exploitation Microsoft Windows 7 permet d'exécuter des applications crées spécifiquement pour Windows XP en faisant tourner ce dernier comme un système invité. Il n'est pas rare non plus de trouver pour toute console de jeux, présente dans le commerce depuis plus d'une dizaine d'années, son équivalent logiciel permettant de jouer à des jeux prévus pour elle sur un ordinateur. Les systèmes virtualisés permettent aussi une plus grande facilité de distribution comme, par exemple, avec la machine virtuelle JAVA (JVM) [14] dont le slogan est *write once, run anywhere*. En fin, la virtualisation est un outil primordial pour la mise en place d'environnements de test. Les développeurs logiciel qui souhaitent rendre leur produit compatible avec de nombreuses plateformes, peuvent utiliser la virtualisation pour facilement, de façon peu coûteuse et sans risque, tester son bon fonctionnement avant de la mettre en production.

Nous faisons ici la distinction entre deux technologies de virtualisation. Les machines virtuelles et les conteneurs. Les machines virtuelles fonctionnent sous la supervision d'un hyperviseur qui a pour but de faire le lien entre les ressources physique et la VM. Ils sont divisés en deux familles, les hyperviseurs de type 1 et 2. Ceux de type 1, tels que Hyper-V

[15], VMWare ESX [16] ou Xen [17], interagissent directement avec la couche physique et n'ont pas besoin de fonctionner sur un système d'exploitation. Ceux de type 2, quant à eux, nécessitent de s'exécuter sur le système d'exploitation hôte. On peut compter parmi eux les hyperviseurs KVM [18] et Oracle VirtualBox [19]. La technologie des conteneurs sera abordée à la section 2.2.5.

2.2.1 Paravirtualisation et virtualisation matérielle

Devant l'utilisation croissante de systèmes virtualisés, il a semblé nécessaire de chercher à les optimiser pour diminuer le surcoût induit par la couche de virtualisation. Le concept, popularisé par l'hyperviseur Xen, revient à permettre une coopération entre les systèmes invité et hôte pour autoriser le système invité à utiliser directement une ressource physique. Cependant cette technique nécessite la modification du système invité. Elle pose peu de problèmes pour des systèmes d'exploitation libres mais devient contraignante pour les systèmes propriétaires dont les développeurs doivent implémenter eux-mêmes la solution.

La virtualisation assistée par matériel est une méthode qui permet d'augmenter les performances de la virtualisation sans avoir besoin de modifier le système invité. Introduite par Intel et AMD avec respectivement leurs extensions Intel-VT [20] et AMD-V [21], cette technique met en place l'utilisation de structures de données spécifiques au contrôle des instances des machines virtuelles, accédées directement par le matériel. Cet accès direct oblige cette structure, aussi appelée VMCS, à dépendre de l'architecture utilisée. Par conséquent, les hyperviseurs voulant bénéficier de ces extensions doivent s'adapter au fabricant ciblé. Cette technique est exploitée dans nos travaux. Comme il nous est nécessaire de détecter quand une machine virtuelle est en train d'exécuter son code, nous utilisons les deux nouveaux modes introduits par la virtulisation matérielle pour arriver à nos fins. Ces modes sont le mode invité, indiquant que le système invité s'exécute, et le mode hôte, indiquant que c'est le code de l'hyperviseur qui s'exécute. Notre stratégie sera d'ailleurs étendue pour continuer à fonctionner même dans le cas de machines virtuelles imbriquées.

2.2.2 The Turtles project

Le Turtles project [22] est une initiative de IBM Research et IBM Linux Technology Center. Elle démarre du constat que les utilisateurs de machines virtuelles ne s'arrêtent plus à un seul niveau de virtualisation mais commencent à utiliser des VMs comme hyperviseurs pour pouvoir lancer de nouveaux des VMs. C'est ce que l'on appelle des machines virtuelles imbriquées. On a déjà parlé du système d'exploitation Microsoft Windows 7 faisant tourner Windows XP dans une VM. Si Windows 7 est déjà lancé en tant qu'une VM, alors son

instance de Windows XP devient une VM imbriquée. Plus généralement, l'utilisation de VMs imbriquées a le potentiel de développer le marché de l'infrastructure en tant que service (IaaS), où un fournisseur pourrait donner la possibilité à ses utilisateurs de choisir ses hyperviseurs lancés en tant que machine virtuelle. Ainsi, l'utilisateur pourrait gérer ses machines virtuelles en sélectionnant l'hyperviseur de son choix. On peut aussi espérer la possibilité de migrer un hyperviseur vers un autre nœud du réseau avec l'intégralité des VMs qu'il contient.

Le problème est qu'à ce jour la virtualisation matérielle ne supporte pas les VMs imbriquées. Plus précisément, un hyperviseur dans une VM ne peut pas utiliser la virtualisation matérielle pour exécuter ses propres VMs. L'objectif à atteindre est donc de se servir de la virtualisation matérielle existante pour pouvoir faire tourner des VMs imbriquées avec entre 6 et 8% de surcoût par rapport à une VM s'exécutant directement sur une machine physique. Fondamentalement, leur méthode consiste à multiplexer les différents niveaux de virtualisation sur le niveau supportant la virtualisation matérielle. Si on appelle le niveau de l'hôte L_0, celui de l'hyperviseur virtualisé L_1 et celui de la VM imbriquée L_2, alors le principe revient à transférer le lancement d'une VM de L_2, de L_1 vers L_0. Quand une instruction nécessitant l'intervention d'un hyperviseur est rencontrée, c'est donc à chaque fois L_0 qui l'intercepte et la transmet à l'hyperviseur d'un autre niveau au besoin. La figure 2.2 résume ce principe.

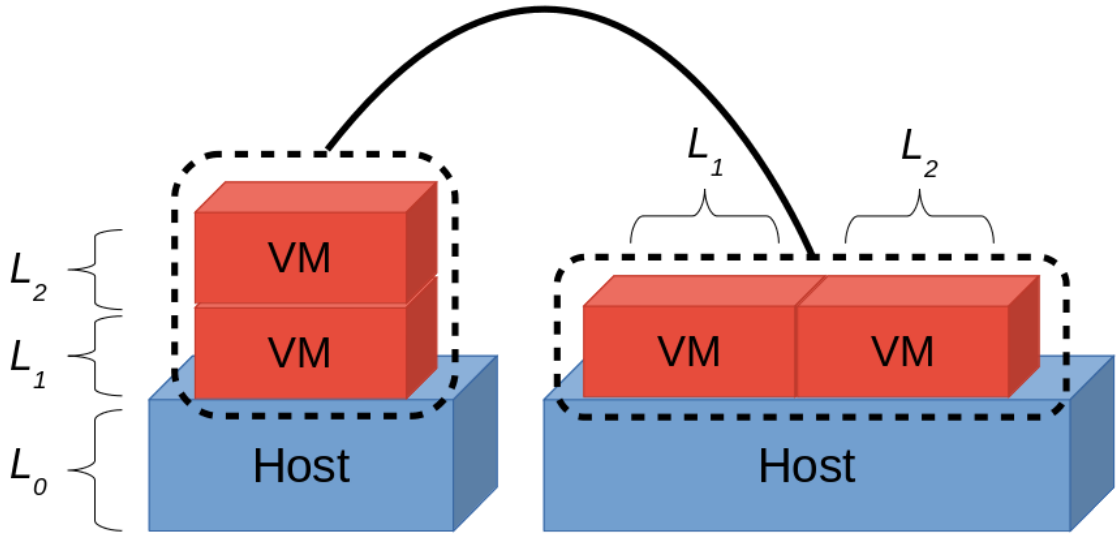

Figure 2.2 Méthode du multiplexage présentée dans le Turtles project.

La figure 2.3 montre que, dans ce cas, il n'y a non pas une VMCS associée pour chaque

VM, mais deux pour chaque VM de L_2. Cette technique, appelée *VMCS Shadowing*, permet à L_0 de directement utiliser la VMCS d'une VM de L_2. Cela n'aurait pas été possible avec celle crée par L_1, car elle n'est pas valide pour l'environnement de L_0.

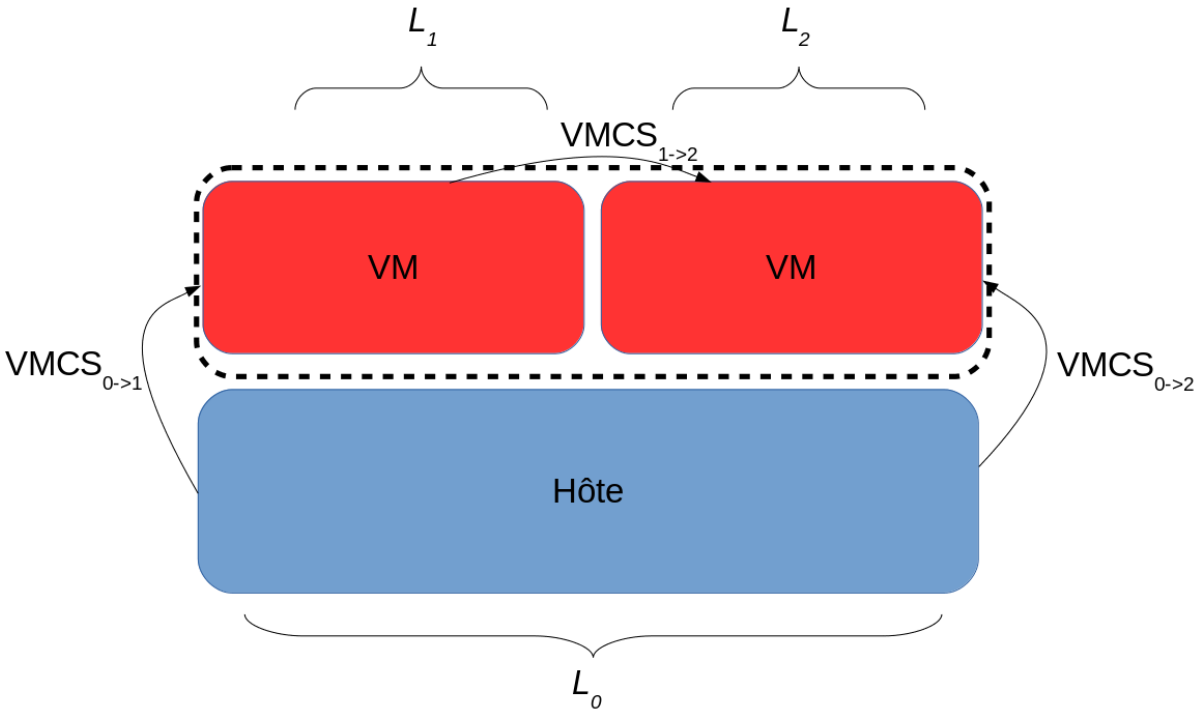

Figure 2.3 VMCS shadowing introduit par le Turtles project.

La gestion optimale de la mémoire pour des VMs imbriquées est aussi un défi. Pour des VMs directement sur l'hôte, le système invité possède une table de pages virtuelles qui doivent être traduites en adresses physiques, dans son propre contexte, pour être enfin converties en adresses physiques sur l'hôte. Une VM imbriquée nécessite donc un troisième niveau de traduction. Cependant le matériel actuel n'est prévu pour qu'au plus deux niveaux de traduction [23]. Dans l'utilisation de VM sans imbrication, pour traduire une adresse virtuelle du système invité, le processeur utilise en premier la table de pages invitée disponible (GPT), puis la seconde table pour convertir une adresse physique de l'invité en adresse physique de l'hôte. Cette seconde table est appelée EPT par Intel et RVI chez AMD. Le Turtles project introduit donc le principe de *multi-dimensional paging* ou pagination multi-dimensionelle, pour pouvoir multiplexer les trois tables de traduction sur les deux disponibles physiquement. Le principe est d'utiliser tout d'abord la GPT pour traduire une adresse virtuelle de L_2 en une adresse physique de L_2, puis les deux tables suivantes sont compressées pour n'utiliser

qu'une seule EPT. Cela permet à L_0 d'utiliser L_2 directement avec cette dernière. Ce principe est illustré par la figure 2.4.

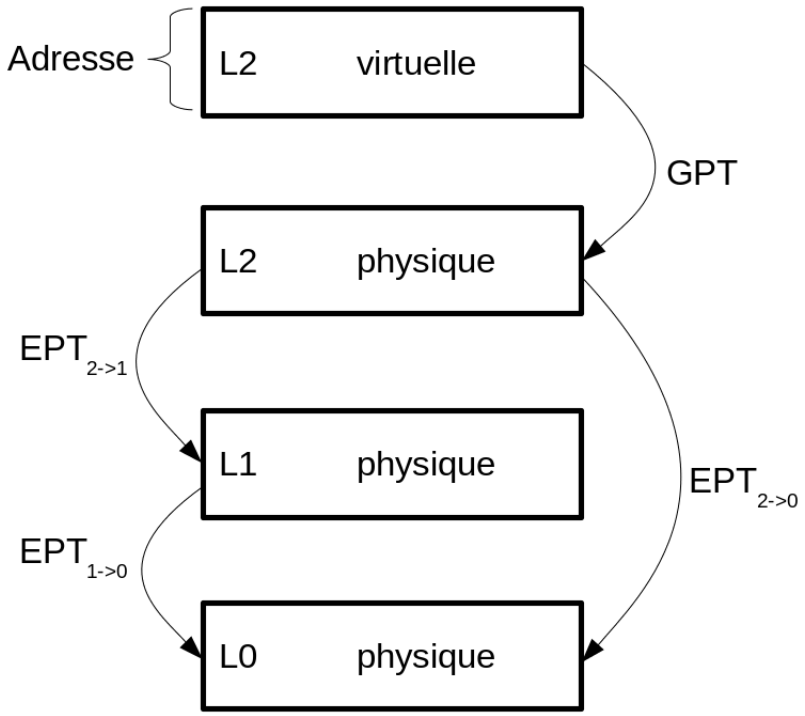

Figure 2.4 Pagination multi-dimensionelle introduite par le Turtles project.

La gestion des entrées/sorties se fait habituellement de trois façons différentes. L'hôte peut émuler complètement un appareil permettant à l'invité de l'utiliser directement, sans modifier ses pilotes, le système invité peut être modifié et profiter de la paravirtualisation, ou bien l'hôte peut directement assigner un matériel réel à l'invité. Le Turtles project a permis à la plupart des combinaisons réalisables, avec deux niveaux de virtualisation, d'être implémentées.

2.2.3 OpenStack

OpenStack [24] est un logiciel libre, initié par la NASA [25] et Rackspace Cloud [26] en 2010, ayant pour principal but de pouvoir offrir à toute organisation la possibilité de profiter des technologies de l'infonuagique avec des équipements standards. Depuis, de nombreuses entreprises ont rejoint le projet dans un but de promotion et de développement. Les principales caractéristiques d'OpenStack sont :

— son adaptabilité : prouvée par son utilisation majeure à travers le monde et la possibilité de déployer jusqu'à soixante millions de machines virtuelles sur un million de machines physiques,
— sa compatibilité et sa flexibilité : lui permettant de supporter de nombreuses solutions de virtualisation telles que KVM, LXC, ou Xen,
— sa caractéristique de logiciel libre : permettant d'être facilement modifiable et adapté en fonction des besoins caractéristiques des compagnies, donnant aussi la possibilité à quiconque de contribuer au projet.

OpenStack a une architecture modulaire dont chacun des sous-ensembles est une brique dédiée au bon fonctionnement du nuage.

Nova

Nova est l'élément contrôlant les nœuds de calcul du nuage. Pour réaliser cela, il communique directement avec les hyperviseurs des machines virtuelles ou par l'intermédiaire d'outils de gestion de la virtualisation tels que la libvirt. Il permet le démarrage, la réinitialisation, la suspension et l'arrêt des instances, de même que le contrôle des accès, des quotas ainsi que du débit. Par ailleurs, il permet également d'utiliser une mise en cache locale des images pour supporter un démarrage plus rapide des instances.

Glance

Glance est le gestionnaire d'images d'OpenStack. C'est lui qui stocke, fournit, ajoute et supprime les images utilisables. En plus de cela, il offre des informations à propos des images qu'il contient par l'intermédiaire de métadonnées. Il possède une banque d'images déjà construites composée, entre autres, d'Ubuntu, Red Hat, Windows, et supporte différents types d'hyperviseurs comme KVM, VirtualBox ou VMWare. La paravirtualisation est souvent utilisée dans la configuration des images, également modifiées pour permettre leur accès par le réseau.

Swift

Swift est utilisé pour la sauvegarde d'objets de façon redondante et évolutive. Il a été conçu pour le stockage à long terme de grandes quantité de données. Sa caractéristique distribuée est particulièrement adaptée pour éviter une situation de point individuel de défaillance ou *single point of failure*. L'information étant répliquée, la défaillance d'un serveur ou d'un disque entraîne une copie des données vers de nouveaux emplacements.

Neutron

Neutron est la composante chargée de la gestion du réseau. C'est lui qui fournit les adresses IP et permet de modifier la topologie du réseau, connecter les instances entre elles et gérer le trafic via les groupes de sécurité. L'attribution des adresses IP peut être aussi bien statique que dynamique, par l'utilisation d'un service DHCP, avec des adresses pouvant être flottantes, permettant un accès des instances par internet. Ceci est couplé à des services de détection d'intrusion, de pare-feu, de réseau privé virtuel (VPN), et d'équilibrage de charge ou *load balancing*. Son développement est également modulaire et lui autorise l'ajout d'extensions lui permettant d'élargir son support d'équipements de gestion du réseau.

Cinder

Les fichiers locaux d'une instance étant volatils, Cinder permet de créer des disques virtuels sous forme de blocs pour les associer à des instances. Cette technique permet la mise en place de bases de données ainsi que la possibilité pour les instances d'avoir un accès bas niveau aux périphériques de stockage. Enfin, Cinder peut créer des instantanés des blocs gérés pour pouvoir les restaurer au besoin.

Horizon

Horizon est l'interface graphique d'OpenStack, offrant une alternative à l'accès en ligne de commande ou par programme. C'est une application web servant de tableau de bord aux utilisateurs pour la gestion de leur nuage. C'est à partir de cette interface que peuvent être gérés de nombreux paramètres liés aux instances, le réseau, le stockage, les images ou les usagers. De plus, elle permet l'affichage de différentes métriques et statistiques pour surveiller l'état des services et des objets. Horizon est très souvent réutilisé par des entreprises pour ajouter de nouvelles métriques ou même des méthodes de facturation.

Keystone

Keystone est responsable du répertoire des usagers, de l'authentification par mot de passe ou par jetons, des politiques d'accès ainsi que de l'exposition des différents services. Tout cela passe par Keystone qui est également compatible avec d'autres services d'annuaire comme LDAP.

Heat

Les paramètres spécifiant les caractéristiques des instances sont gérées par Heat. Pour créer des machines, l'utilisateur doit utiliser des gabarits définissant une recette à suivre pour la création des instances. Ce modèle est fourni à Heat qui ira ensuite déployer l'infrastructure décrite. Enfin, Heat permet de s'appuyer sur des métriques fournies par Ceilometer pour pouvoir modifier dynamiquement la structure du nuage. Par exemple, il est possible de créer automatiquement de nouvelles instances si la charge des nœuds de calcul dépasse un certain seuil. Inversement, il est possible de réduire le nombre d'instances si certaines d'entre elles sont sous-utilisées.

Ceilometer

Ceilometer est un système modulaire servant à la collecte et l'agrégation d'informations sur l'exécution d'OpenStack. Il donne accès à un grand nombre de métriques telles que l'activité des CPUs, le nombre d'accès en lecture ou écriture, la quantité d'accès au réseau, ou le nombre d'instances créées et détruites depuis le lancement d'OpenStack ainsi que leur taux d'utilisation. Ces informations peuvent alors être utilisées pour faire du suivi d'instances, pour déterminer des pics d'utilisation, agir dynamiquement sur machines créées, avec Heat, ou juste avoir une vue d'ensemble du nuage avec Horizon. Enfin, les informations recueillies permettent aussi de mettre en place des méthodes de facturation, relativement au temps CPU utilisé, l'utilisation du réseau, le nombre d'instances utilisées ou bien d'autres critères des compagnies.

Sahara

Sahara, précédemment appelé Savanna, permet d'utiliser des clusters Hadoop [27] par la spécification de paramètres spécifiques au cluster ciblé. En réutilisant des services comme Horizon, Glance, Nova, Swift et Keystone, Sahara permet le déploiement rapide du cluster décrit. De plus, il est aussi possible d'adapter le cluster à la charge qui lui est soumise en ajoutant et enlevant des nœuds à la demande.

Trove

L'installation, la gestion et l'accès à des services de base de données relationnelle et NoSQL se font par l'intermédiaire de Trove. Cet accès se fait grâce à des appels à distance (RPC). Suite à la demande d'un client reçue par un agent, l'agent en question soumet la demande à la base de données puis renvoie la réponse au client. Trove supporte un large ensemble de

base de données telles que Cassandra, Couchebase, Mongodb, MySQL, Percona, PostgreSQL et Redis.

Marconi

Marconi est le service de gestion de messages. Il est chargé des queues de messages, de la publication d'abonnements ainsi que des notifications. Ses fonctionnalités sont notamment utiles pour la distribution de tâches entre les nœuds, la collecte des données ainsi que l'envoi de commandes à différents destinataires.

Ces modules sont complétés par d'autres services déjà intégrés à OpenStack, tels que ceux de gestion de clés (Barbical), de gestion des DNS (Designate), de *Bare Metal provisioning* (Ironic), de gestion des systèmes de fichiers partagés (Manila), de gestion des conteneurs (Magnum) et d'intergiciel à la demande (Zaqar). La figure 2.5 montre un résumé des liens entre les différents modules d'Openstack.

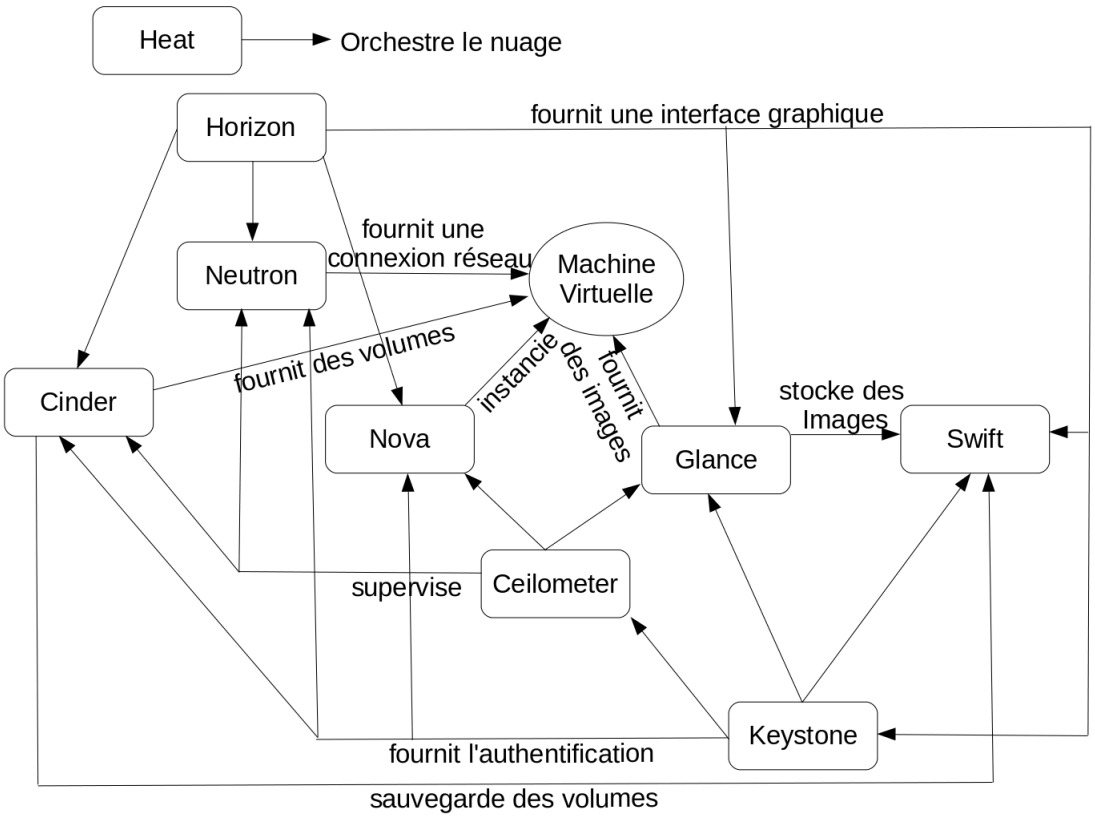

Figure 2.5 Relations entre les différents modules d'OpenStack.

En conclusion, OpenStack est une agrégation de nombreuses entreprises liées par les tech-

nologies de la virtualisation. Même si initialement ses fonctionnalités étaient assez limitées, ses progrès sont rapides, générant une nouvelle version disponible deux fois par an.

2.2.4 Préemption de machine virtuelle

La virtualisation permet entre autres de réduire la sous-utilisation des ressources physiques. Cependant, comme plusieurs systèmes virtualisés peuvent cohabiter sur une même machine, il est possible que ces ressources se retrouvent en surutilisation car en nombre insuffisant par rapport à la demande. Dans le cas des machines virtuelles, un manque de CPU physiques de l'hôte peut mener à une préemption des CPU virtuels (vCPU). Dans cette situation, le vCPU de la VM et par conséquent le fil d'exécution courant, sont stoppés au profit d'un autre processus ayant lui aussi besoin d'utiliser la ressource. Du point de vue de la VM, tout se passe comme si la préemption n'avait pas lieu. Néanmoins, la tâche stoppée mettra plus de temps à terminer. Il est donc utile de se doter d'outils capables de détecter cette préemption, preuve que la configuration du système peut être optimisée.

C'est dans cette optique qu'ont été développés des outils d'analyse et de visualisation par Gebai et al. [28] pour l'hyperviseur KVM. Leurs travaux ont permis la création d'une analyse transformant des traces noyau, effectuées sur un hôte et ses VM, en une vue montrant l'état de chaque vCPU en tout temps. Il est alors possible d'observer si un vCPU est actif, inactif car non utilisé, préempté par l'hyperviseur de sa VM, ou bien préempté par une cause extérieure. La figure 2.6 montre le résultat de l'analyse dans le cas d'une VM avec deux vCPUs autorisés à n'utiliser qu'un seul CPU physique de son hôte. Il en résulte que les tâches de chacun des vCPUs se préemptent mutuellement pour accéder au seul CPU disponible. On peut voir l'alternance des vCPUs préemptés en violet. L'algorithme utilisé est basé sur une machine à état représentant chaque vCPU. La trace noyau de l'hôte permet de détecter les transitions du CPU physique entres ses modes invité et hôte. Couplée à la trace noyau de la VM, il est alors possible de déterminer si un processus était en cours d'exécution lors de la sortie du mode invité et, le cas échéant, de signaler le vCPU comme étant préempté.

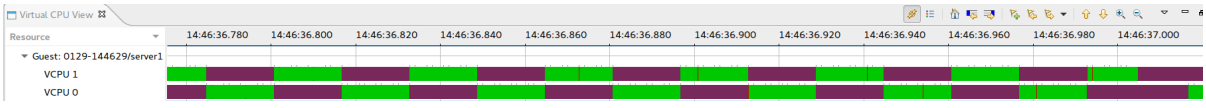

Figure 2.6 Exemple de deux vCPUs se préemptant mutuellement pour la prise d'une ressource commune.

Leurs travaux ont également abouti au développement d'une analyse complémentaire capable de retrouver le flot d'exécution d'un fil d'exécution pour observer les raisons de sa préemption. Cependant le modèle utilisé n'est pas adapté au cas des machines virtuelles imbriquées. C'est un point que nous complétons dans notre étude.

Shao et al. [29] ont proposé un analyseur d'ordonnancement pour l'hyperviseur Xen. Leur analyseur utilise une trace fournie par Xen pour reconstruire l'historique de l'ordonnancement de chaque CPU virtuel. Ainsi, il leur est possible de récupérer plusieurs métriques. Cependant, une trace produite par Xen n'est pas suffisante pour déterminer quel processus aurait pu causer une perturbation dans une VM.

2.2.5 Contextualisation

La contextualisation est une technique de virtualisation permettant de produire des conteneurs. Les conteneurs sont parfois aussi appelés machines virtuelles, mais il faut cependant ne pas les confondre avec les techniques de virtualisation complètes de systèmes d'exploitation. C'est pourquoi il reste préférable de ne pas utiliser le terme VM quand on souhaite désigner un conteneur. Le principe de la contextualisation est de subdiviser les ressources du système, comme les processeurs, la mémoire vive ou les connexions réseau, pour les partager avec les différents conteneurs, tout en leur donnant accès à un seul et même noyau, celui de l'hôte. Les manières de subdiviser les ressources ont été implémentées de diverses façons et portent de multiples noms. On peut, par exemple, citer les jails sous FreeBSD, les zones sous Solaris, mais encore LXC, OpenVZ et Linux VServers sous Linux. C'est LXC qui sera utilisé dans nos travaux car d'une part il est inclus dans le noyau Linux et d'autre part il est grandement utilisé, notamment par l'intermédiaire de Docker et LXD.

LXC permet le contrôle des ressources par l'intermédiaire des groupes de contrôle (cgroups). Les groupes de contrôle lient un ensemble de processus avec des critères qui définissent des limites d'utilisation. Par exemple, il est possible de limiter un groupe de processus à n'utiliser qu'un certain pourcentage d'un sous ensemble de processeurs. Les groupes de contrôles sont organisés similairement aux processus sous Linux. Les groupements peuvent être hiérarchiques, impliquant qu'un sous groupe héritera des limitations de son groupe parent. Cependant, alors qu'il n'existe qu'une seule hiérarchie de processus, il peut en exister simultanément plusieurs pour les groupes de contrôle. Cette organisation est nécessaire car chaque groupe est lié à une ou plusieurs ressources ainsi que ses limitations associées.

La séparation entre processus se fait par l'intermédiaire de ce que l'on a appelle des espaces de nommage ou espaces de nom. Ils sont utilisés pour autoriser différents objets à avoir le même nom dans différents contextes. Dans le même ordre d'idée que deux variables peuvent

avoir le même nom dans deux fonctions différentes, il est possible de créer deux processus avec le même identifiant. Cependant, cet identifiant n'est valable que dans son contexte. En réalité, ces deux processus ont des identifiants réels, différents l'un de l'autre. Suivant le même principe que les identifiants de processus, la couche réseau des conteneurs est elle aussi contextualisée. De ce fait, chaque conteneur a l'illusion de posséder sa propre adresse sur le réseau.

Même si cette technologie apporte l'avantage d'isoler certaines parties du système tout en donnant de bonnes performances car l'utilisation des ressources est faite nativement, elle apporte son lot de contraintes. La flexibilité est sacrifiée au profit de la performance. En effet, comme les conteneurs partagent leur noyau avec leur hôte, ils ne peuvent pas choisir leur système d'exploitation. De plus, comme l'intégralité de l'activité des conteneurs est visible depuis l'hôte, une faiblesse dans les fonctions du noyau permettant à un utilisateur malicieux de s'octroyer plus de droits lui permettrait d'avoir accès aux autres conteneurs ou de perturber leur fonctionnement. Ce principe est illustré dans la figure 2.7, où le conteneur C3 interfère avec les conteneurs C1 et C2. Par ailleurs, LXC, bien qu'inclus dans le noyau Linux depuis la version 2.6.24, ne propose pas encore l'intégralité des fonctionalités proposées par d'autres solutions de contextualisation comme la migration sans arrêt du conteneur.

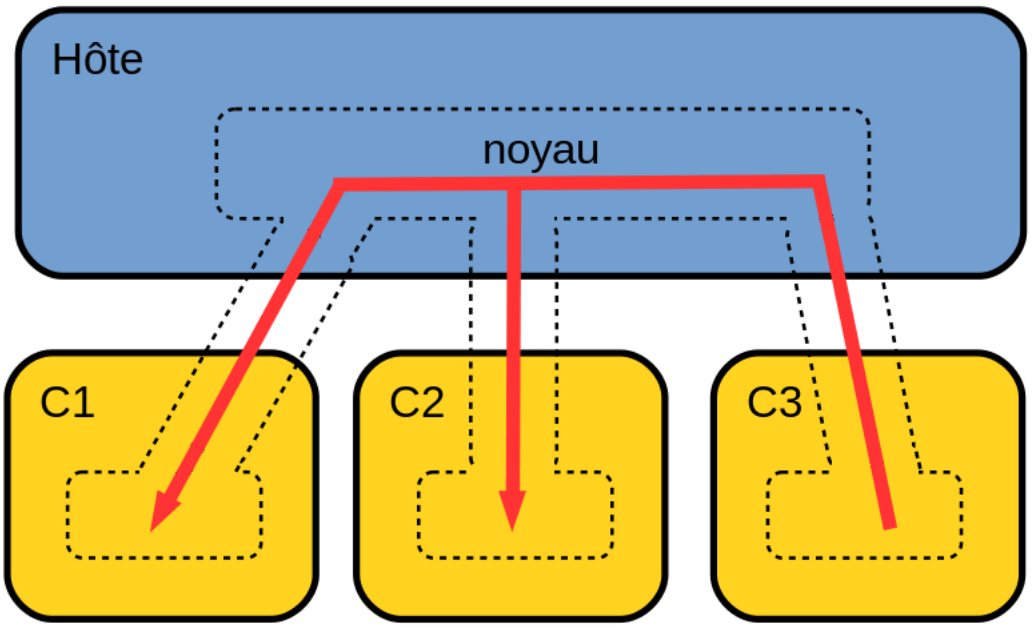

Figure 2.7 Accès ou perturbation de conteneurs par partage du noyau.

Docker

Docker [30] est un logiciel libre initié par dotCloud [31] en 2013. Développé en Go [32], c'est une surcouche logicielle autour des techniques de contextualisation. Son but est de pouvoir facilement créer des encapsulations de logiciels, prêtes à être déployées. Pour cela, Docker ajoute dans cette capsule l'intégralité du contenu nécessaire au fonctionnement de l'application tel que son code ou des outils et librairies système. L'idée principale de Docker est de n'avoir qu'une seule application par conteneur et de la rendre portable. Ainsi, cela permet de se décharger de "l'enfer des dépendances", dont souffrent les applications nécessitant d'autres services. Il arrive souvent que lors du déploiement d'une application, des dépendances se retrouvent manquantes ou entrent en conflit entre elles. Docker propose une solution permettant de ne plus avoir besoin de se soucier de ces problèmes.

LXD

Bien que souvent pris pour un concurrent direct à Docker, LXD [33] se décrit comme une extension de LXC. Projet fondé et dirigé par Canonical Ltd [34] et Ubuntu [35], LXD est un hyperviseur de conteneurs structuré en trois composants qui sont :

— un *system-wide daemon*,
— un client en ligne de commande,
— le plugiciel OpenStack Nova

Le daemon a pour but d'exporter une API REST [36] localement et, si possible, sur le réseau. Le client en ligne de commande est un outil puissant, malgré sa simplicité, pour gérer l'ensemble des conteneurs. Il permet de créer, accéder et stopper les instances, ainsi que d'avoir une vue d'ensemble des conteneurs lancés sur le réseau. Le plugiciel OpenStack permet d'utiliser les instances comme des nœuds de calcul. LXD apporte son lot de fonctionnalités telles qu'un apport en sécurité, par l'intermédiaire de conteneurs non privilégiés et des restrictions d'utilisation de ressources, une mise à l'échelle permettant d'utiliser jusqu'à plusieurs milliers de nœuds de calcul, ainsi que le support de la migration à chaud. LXD n'est pas une réécriture de LXC mais une extension de celui-ci. Il permet d'apporter une meilleure expérience d'utilisation de part sa simplicité ainsi que ses fonctionalités.

2.3 Synchronisation

La synchronisation est un aspect fondamental dans l'analyse de traces de systèmes distribués. Chaque système d'exploitation étant responsable de sa propre gestion du temps, la négliger rendrait impossible la corrélation d'évènements provenant de traces différentes. Les

systèmes virtualisés, bien que s'éxécutant sur leur hôte, n'échappent pas à ce besoin.

L'algorithme de base est celui proposé par Jabbarifar [37] intitulé "*Fully incremental convex hull synchronisation algorithm*". Son fonctionnement repose sur l'appariement d'évènements entre les différentes traces. Savoir qu'un évènement unique a de la trace A, enregistrée sur M_1, arrive forcément avant un autre évènement b de la trace B, enregistrée sur M_2, impose une borne inférieure sur la valeur de l'estampille de temps associée à l'évènement b. Une autre relation d'ordre cette fois de B vers A permet d'obtenir à nouveau une borne inférieure mais pour un évènement de A. Ces couples d'évènements forment des points représentés dans un graphe ayant pour abscisse et ordonnée les temps relatifs à M_1 et M_2. Les paires représentant le sens M_1 vers M_2 verront leurs points apparaître dans la partie haute et gauche du graphe tandis que celles représentant le sens M_2 vers M_1 apparaissent dans la partie basse et droite. La droite pour l'estimation de synchronisation se situe entre ces deux nuages. Pour la calculer, il faut utiliser les paires ayant le minimum de latence. Ces paires sont trouvées en calculant l'enveloppe convexe des deux nuages de points. Cette méthode rejette donc certains points, la rendant de ce fait plus précise qu'une régression linéaire. La droite recherchée est alors la bissectrice entre les droites ayant respectivement un coefficient directeur maximal et minimal, sans couper les deux enveloppes convexes. La figure 2.8 montre ces nuages de points ainsi que la droite pour l'estimation de synchronisation.

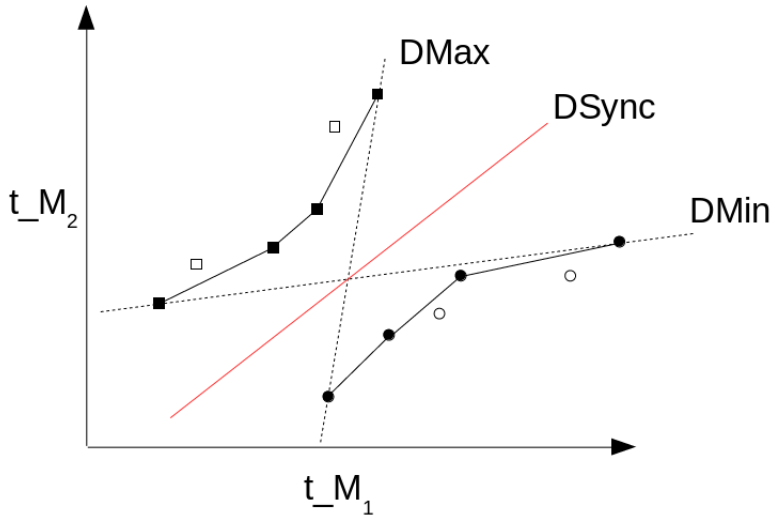

Figure 2.8 Détermination d'une formule de synchronisation par la méthode des enveloppes convexes.

Usuellement, le trafic de paquets TCP sert à générer des évènements à apparier. Ces évènements correspondent parfaitement aux critères de l'algorithme, d'une part parce qu'ils

sont identifiés de façon unique, et d'autre part parce que le départ d'un paquet survient forcément avant son arrivée. Cette approche peut donc s'appliquer aux cas de systèmes distribués, communiquant entre eux par le réseau.

Cependant, cette communication n'existe pas toujours entre un hôte et ses machines virtuelles. Pour palier à ce problème, Gebai et al. ont proposé d'utiliser les hypercalls pour pouvoir générer les évènements nécessaires à la synchronisation. Un hypercall est un moyen de communication d'une VM vers son hôte qui peut être utilisé pour transmettre des valeurs numériques. Ici, cette valeur sera donc exploitée pour transmettre un numéro de séquence, identifiant de façon unique l'hypercall. L'hypercall est déclenché par un module noyau chargé sur le système invité. La génération de l'évènement sur l'hôte est, quant à elle, gérée par un autre module noyau spécifique au système hôte. La figure 2.9 résume l'ordre de génération des évènements utilisés pour la synchronisation d'une VM avec son hôte.

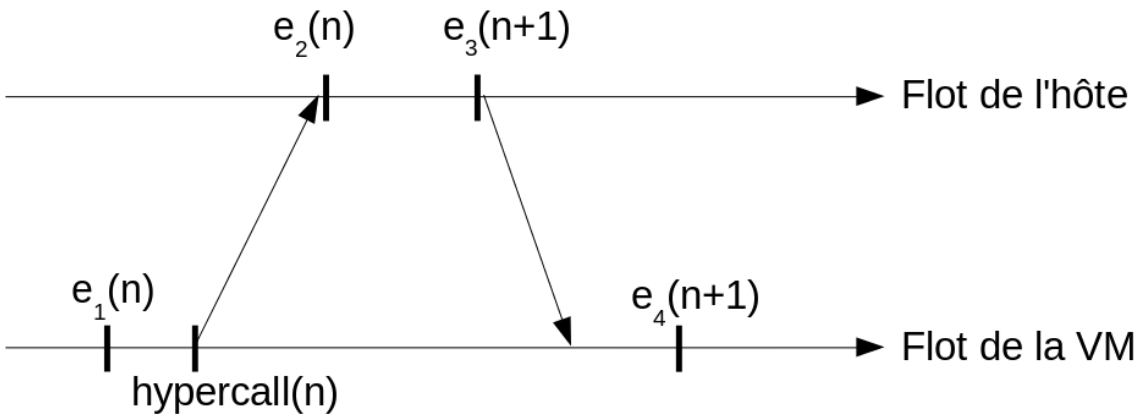

Figure 2.9 Suite d'évènements entre une VM et son hôte pour la synchronisation de traces.

Comme un hypercall généré par une VM est forcément pris en charge par son hôte, cette méthode peut aussi s'appliquer dans le cas des machines virtuelles imbriquées. Les modules noyau sont alors chargés sur les VMs de L_1 et L_2. Il en résulte qu'une trace de L_0 sera synchronisée avec celle de L_1, qui elle même le sera avec L_2.

2.4 Visualisation de traces

L'analyse de trace se termine presque toujours par l'utilisation d'un outil de visualisation autre que textuel. Celui-ci se doit de respecter certains critères car il est directement manipulé

par les utilisateurs. Même l'analyse la plus complète qui soit devient obsolète si la vue lui étant associée ne l'exploite pas convenablement. La vue doit être adaptée aux besoins de l'utilisateur. Par exemple, si on possède une vue représentant l'exécution d'un système dans son ensemble et que l'utilisateur connaît le nom d'un processus qu'il veut observer, une méthode de recherche par nom doit lui être fournie. S'il ignore son nom mais qu'il sait quand il a débuté, il faut alors lui donner la possibilité d'observer le système à l'instant souhaité. S'il ne sait rien de ce qu'il veut observer mais qu'il remarque un comportement étrange dans la vue, il faut lui permettre d'en savoir plus sur ce qu'il observe. Un des concepts de base à appliquer pour la conception d'outils de visualisation est le *mantra* de Shneidermann [38] : *Overview first, zoom and filter, then details-on-demand*. Il recommande donc une visualisation d'abord d'ensemble pour aller vers le détail, par l'intermédiaire du zoom et du filtrage.

Même si les écrans actuellement sur le marché peuvent afficher plusieurs millions de pixels, cela n'est pas une raison pour les développeurs d'essayer d'afficher simultanément des millions d'informations. La figure 2.10 montre une tentative d'affichage des transitions entre processus pendant une durée de quelques secondes. La cadence d'exécution d'un ordonnanceur fait que des milliers de changements de contextes peuvent survenir, rendant leur visualisation dans l'ensemble peu lisible.

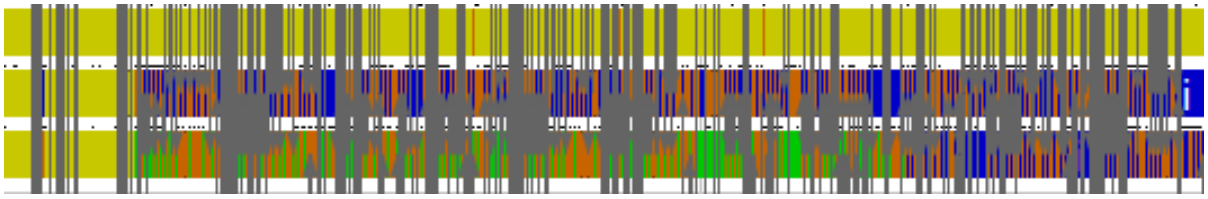

Figure 2.10 Affichage des transitions entre processus dans la vue Control Flow de Trace Compass.

Quoi qu'il en soit, même si on dispose d'une technologie pour afficher de façon lisible l'intégralité des informations disponibles, un être humain ne pourrait pas traiter toute l'information présentée. La limitation des capacités humaines est donc à prendre en compte pour la visualisation de données complexes.

2.4.1 Ocelotl

Ce constat impose donc l'utilisation de techniques servant à réduire la quantité d'informations à afficher et, plus généralement, qui respectent le *mantra* de Shneidermann. C'est ce

genre d'objectif qui est relevé dans la thèse de Dosimont [39], dont les travaux proposent une solution satisfaisant chaque étape du *mantra*. Pour l'étape d'overview, il prend en compte les aspects déjà évoqués, tels que le support d'affichage ou l'utilisateur, pour mettre au point une technique d'agrégation étendant les travaux de Lamarche-Perrin [40]. Cette technique s'occupe de réduire la complexité d'un système avant d'essayer de le visualiser, tout en laissant une flexibilité à l'utilisateur pour contrôler la perte d'information induite. Dans le cadre de cette thèse a été développé Ocelotl [41].

Ocelotl est un outil d'analyse et de visualisation de trace produisant une vue d'ensemble dont il est possible de contrôler le niveau d'agrégation des données. Deux vues sont actuellement disponibles. La première a une approche temporelle et l'agrégation n'est donc appliquée que sur cette composante. Elle a pour principal objectif de s'employer dans des cas où l'application tracée a un comportement homogène. La seconde vue a une approche spatio temporelle. L'agrégation est alors appliquée à la fois pour la variable temporelle et pour les composants tracés. Cette seconde vue s'utilise surtout pour diagnostiquer des baisses de performances dues à certaines ressources. Les figures 2.11 et 2.12 montrent respectivement la vue avec agrégation temporelle avec un fort et faible paramètre d'agrégation.

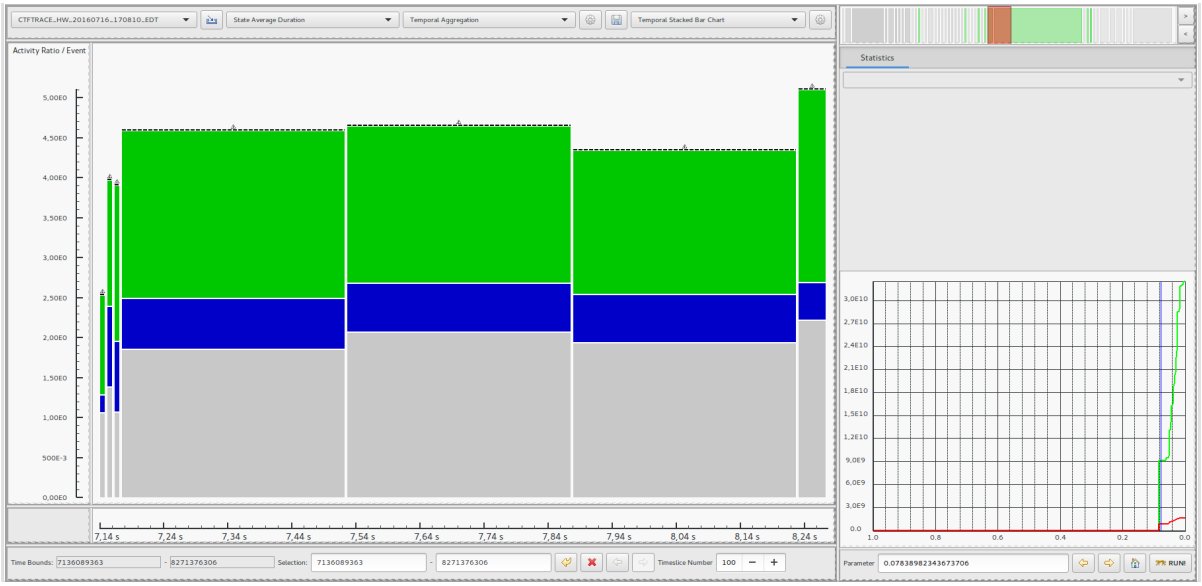

Figure 2.11 Agrégation temporelle dans Ocelotl avec un fort paramètre.

La vue a été paramétrée pour mettre en évidence la proportion moyenne des états des processeurs. Les états espace utilisateur, espace noyau et inactif sont respectivement représentés en vert, bleu et gris. La sélection du paramètre d'agrégation se fait par l'intermédiaire

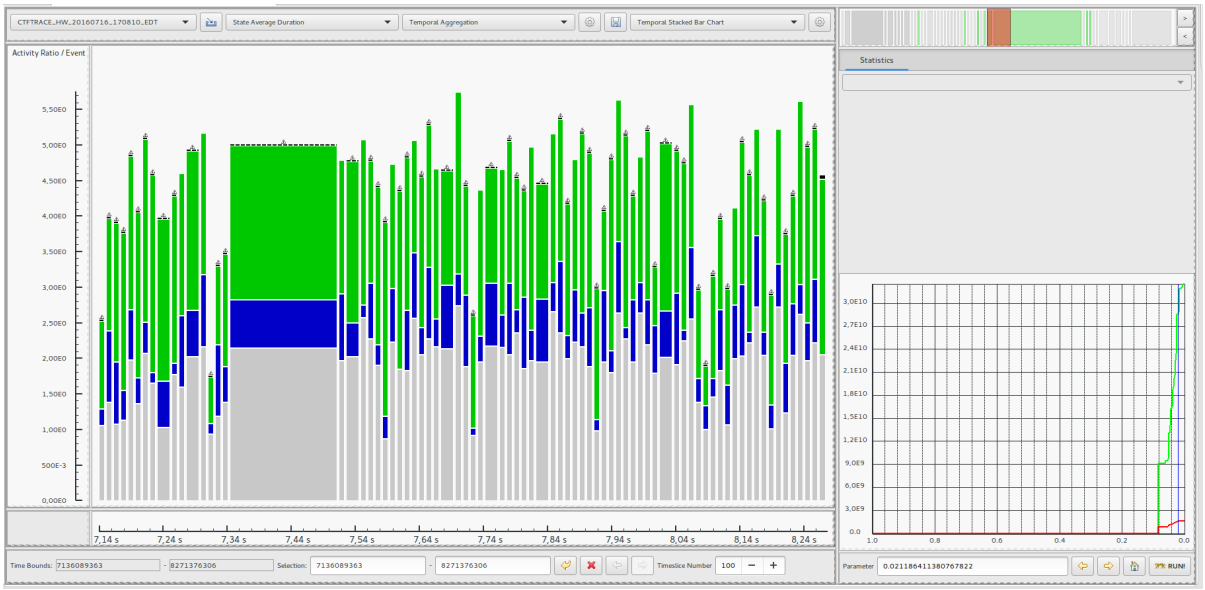

Figure 2.12 Agrégation temporelle dans Ocelotl avec un faible paramètre.

du graphe situé à droite de la vue. La courbe rouge représente la quantité d'information gagnée lors de l'augmentation du nombre d'agrégats, appelé aussi le gain d'information, tandis que la courbe verte représente la quantité d'information nécessaire pour afficher les agrégations et leurs valeurs. Ces deux métriques sont respectivement issues de la divergence de Kullback-Leilber [42] et de l'entropie de Shannon [43].

2.4.2 Trace Compass

Trace Compass [44] est un logiciel libre spécialisée dans l'analyse et la visualisation de traces. Son principal objectif est de fournir des graphes, de vues et des métriques utiles à la détection et au diagnostique de problèmes de performance dans les systèmes tracés. Il supporte de nombreux formats de trace comme les formats CTF, Best Trace Format (BTF), libpcap et même des formats customisés si on lui fournit un analyseur XML correspondant.

Une des analyses les plus importantes de Trace Compass est la *Kernel Analysis* ou l'analyse noyau. Elle s'utilise sur une trace noyau et crée un arbre d'attributs résumant le plus fidèlement possible l'état du système, pendant toute la durée de la trace. Pour faire cela, elle traite les évènements comme les changements de contexte, les entrées et sorties d'appels système ou bien les débuts et fins d'interruptions matérielles et logicielles. Trace Compass fournit aussi une vue permettant d'observer l'état d'un arbre d'attributs. Cette vue, montrée

par la figure 2.13, permet aux développeurs d'analyse de vérifier si la population de leur arbre d'attributs s'est faite de façon adéquate. Dans cet exemple, on peut observer qu'à l'instant sélectionné, le processus 2549 était en cours d'exécution sur le CPU 1.

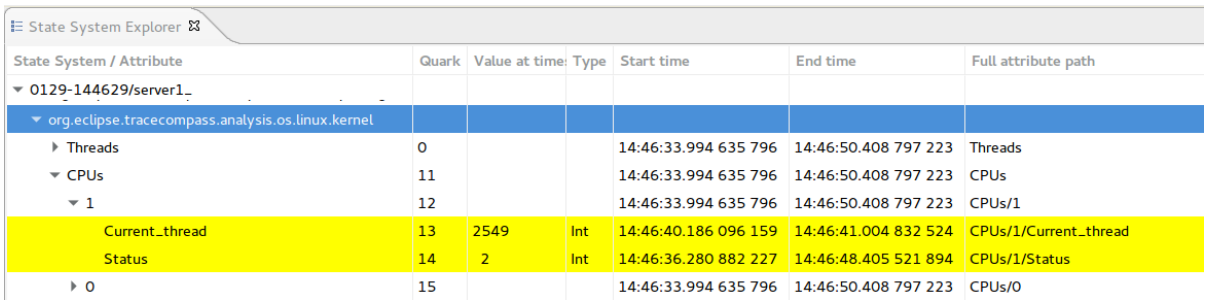

Figure 2.13 Vue State System Explorer de Trace Compass.

Trace Compass fournit des vues de différents types, comme des histogrammes, des diagrammes XY, de Gantt, circulaires, ainsi que des vues statistiques. La figure 2.14 montre la vue brute de Trace Compass pour visualiser les évènements d'une trace. Elle est comparable à la sortie de l'utilitaire babeltrace.

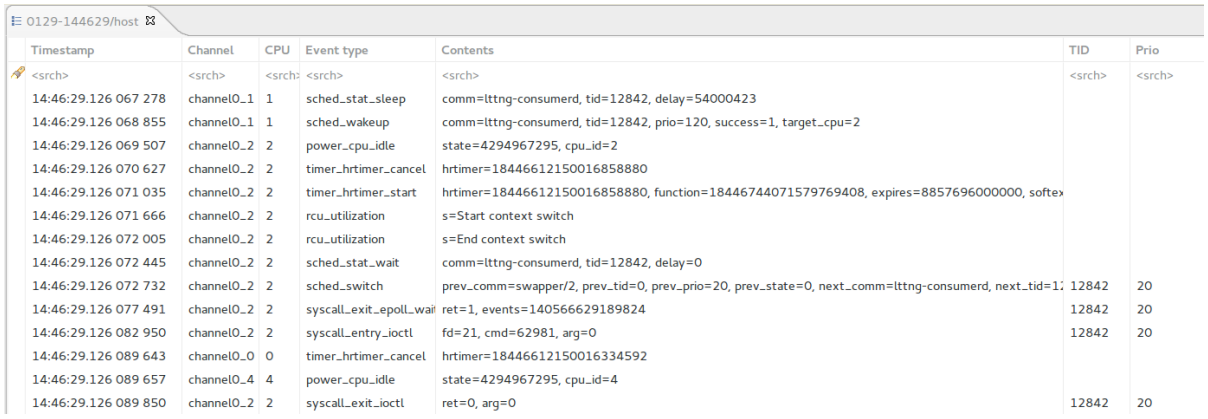

Figure 2.14 Visualisation brute des évènements dans Trace Compass.

La vue Resources est de type graphe temporel. Elle exploite l'arbre d'attributs généré par l'analyse noyau pour afficher une entrée par CPU, montrant à tout instant son état. La figure 2.15 donne un apperçu de cette vue. On peut y voir des processus s'exécutant sur des CPUs en espace utilisateur, en vert, et en espace noyau, en bleu. Les moments où le CPU est inactif

sont signalés en gris. La vue affiche également en rouge les périodes où des évènements ont été perdus.

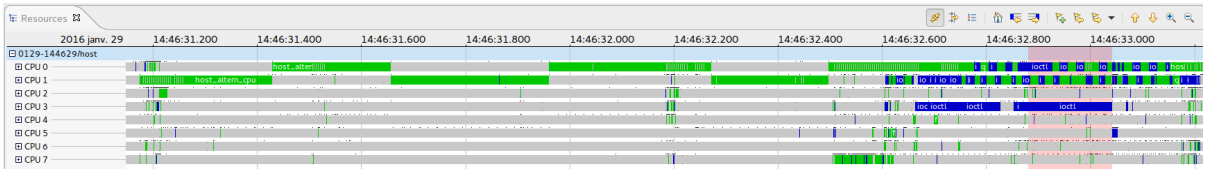

Figure 2.15 Vue Resources de Trace Compass.

Dans ce type de vue, l'agrégation est gérée automatiquement pour afficher les informations par rapport à la résolution de l'écran. La figure 2.16 montre que lorsque des états sont trop petits pour être représentés, ils sont remplacés par une marque noire au dessus de leur position. L'inconvénient de cette méthode est qu'elle n'indique pas ce qui a été agrégé.

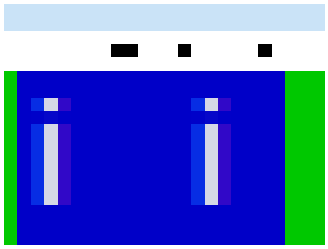

Figure 2.16 Méthode d'agrégation employée dans Trace Compass

Trace Compass, tout comme Ocelotl, est constitué de plugiciels Eclipse. Cette architecture permet de rapidement créer des extensions à la liste d'analyses et de vues disponibles en exploitant les points d'extension d'Eclipse. C'est par cet intermédiaire qu'ont été développées l'analyse et la vue présentées dans nos travaux.

2.4.3 KernelShark

KernelShark est un outil graphique servant à visualiser les traces générées par trace-cmd. Il offre donc une alternative à la lecture brute des traces de Ftrace. Lorsqu'une trace est chargée, on peut alors observer deux zones, respectivement semblables aux graphes temporels et à la vue brute de Trace Compass. La zone graphique permet d'afficher des données relatives aux CPUs ou aux tâches tracées. Pour les entrées correspondant aux CPUs, chaque couleur

représente un fil d'exécution différent. Inversement, pour chaque entrée d'un fil d'exécution, chaque couleur représente un CPU différent. Il devient alors assez facile d'observer l'exécution d'un processus sur les différents CPUs. La figure 2.17 montre un exemple de visualisation de trace avec KerneShark.

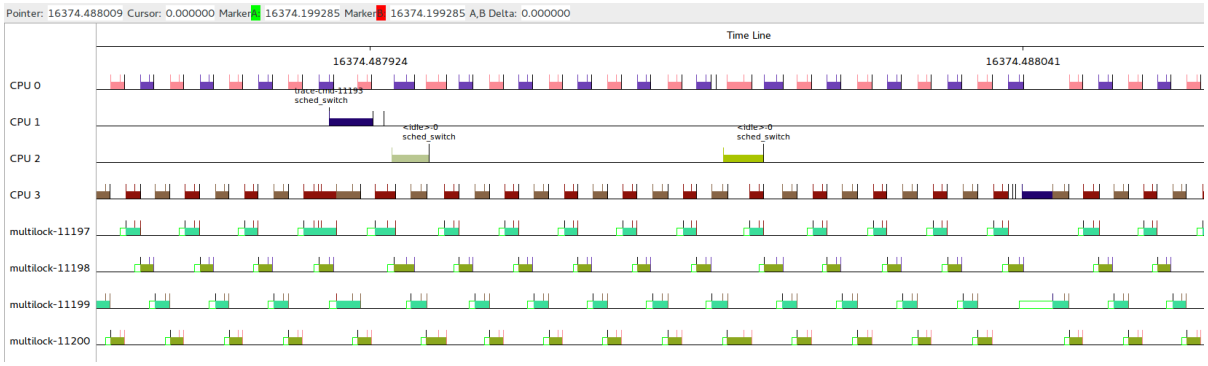

Figure 2.17 Graphe temporel de KernelShark.

La trace visualisée a été produite avec la commande suivante :

```
# trace-cmd record -e 'sched_wakeup*' -e sched_switch -e 'sched_migrate*' relay
```

Elle permet de n'activer que les évènements liés aux changements de contexte, au réveil et à la migration des tâches. Le programme appelé crée quatre tâches qui ne peuvent s'exécuter qu'un bref instant avant de réveiller la tâche suivante. On peut donc observer dans KernelShark les quatre tâches prendre la main sur les CPUs 0 et 3, tandis que les entrées par fil d'exécution montrent bien que chaque tâche attend d'être réveillée par la précédente.

CHAPITRE 3

MÉTHODOLOGIE

L'objectif de ce chapitre est de s'attarder sur les motivations et les choix de conception qui ont mené à la rédaction du chapitre 4. Il est aussi l'occasion de faire un bref récapitulatif des contributions apportées par ces travaux.

3.1 Fusion de systèmes virtualisés

L'idée de fusionner les informations venant des différents systèmes virtualisés pour les faire apparaître sur un seul niveau, a été principalement inspirée les travaux de Gebai et al. Une partie de leurs travaux se concentre sur la visualisation des états de CPUs virtuels à travers le temps, permettant ainsi de savoir s'ils ont été préemptés. Le concept introduit dans ce mémoire est d'aller à contre courant de cette visualisation de vCPUs, pour se concentrer cette fois sur les CPUs physiques de l'hôte. Notre analyse et vue se veulent donc complémentaires dans la façon d'observer des systèmes virtualisés sur leur hôte.

3.1.1 Mise en place de la fusion

L'idée de fusionner les traces d'exécution de plusieurs systèmes a demandé la mise en place d'une stratégie efficace permettant la réalisation de notre objectif.

Fusion a posteriori

Trace Compass intègre une analyse de traces noyau créant un arbre d'attributs, résumant l'état de chaque CPU de chaque système tracé en tout temps. Cependant cette analyse est propre à la trace analysée et ne prend pas en compte les corrélations entre les différentes traces. Malgré tout, chaque arbre créé possède les informations voulues pour être représentées sur un seul niveau. Fusionner l'arbre résultant de l'analyse de l'hôte avec ceux des systèmes virtualisés peut sembler une option. Cette possibilité a été écartée car les informations de corrélation ont déjà été perdues suite à l'analyse noyau.

Analyse en série

Il ne faut donc pas utiliser telle quelle l'analyse noyau, mais bien créer une analyse à part qui prendra en considération les informations de corrélation permettant de lier les traces entre elles. L'analyse en série revient à appliquer cette nouvelle analyse à une des traces, pour créer l'arbre d'attributs incluant les informations de corrélation, avant d'analyser la trace suivante. La seconde analyse devra donc modifier le précédent arbre, ou bien en créer entièrement un nouveau. Cependant, cette méthode est très peu adaptée aux arbres d'attributs car, une fois créés, il devient difficile de les modifier. De plus, créer entièrement un nouvel arbre fait croître la complexité de l'analyse. Beaucoup de temps serait perdu à copier des informations de l'ancien vers le nouvel arbre, surtout si le processus doit être répété autant de fois qu'il y a de traces.

Méthode retenue

Finalement, la méthode retenue est de lire toutes les traces simultanément en listant tous les évènements par ordre d'apparition. Cette méthode semble la plus naturelle et adaptée à notre objectif. En effet, comme on désire représenter l'entière activité des systèmes virtualisés comme s'ils s'étaient exécutés directement sur l'hôte, il est intuitif de créer une analyse allant dans ce sens, traitant les évènements comme s'ils venaient tous d'une seule et même grande trace. Cela apporte l'avantage de ne construire qu'un seul arbre d'attributs.

3.2 Contributions additionnelles

Les travaux présentés ici ont été l'occasion de contribuer à la communauté scientifique. En plus des nouveaux algorithmes et techniques proposés dans l'article du chapitre 4, quelques contributions supplémentaires peuvent être mentionnées. Une contribution intéressante à Trace Compass a été l'ajout d'une fonctionnalité pour permettre la sélection de la trace servant de référence pour la synchronisation. À l'origine, la trace choisie était la première dans l'ordre lexicographique par rapport à l'identifiant aléatoire de celle-ci. Ce choix peut paraître peu important pour des systèmes distribués où l'horloge de chaque machine peut être considérée comme étant précise. Cependant ce n'est pas tout le temps le cas avec des systèmes virtualisés où il devient préférable de choisir l'hôte comme référence. Quelques correctifs pour Trace Compass ont été produits, corrigeant des bogues mineurs, par exemple le rapport d'un bogue dans LTTng empêchant le chargement d'un module noyau nécessaire au traçage d'évènements liés à l'hyperviseur KVM.

Finalement, ces travaux ont mené à la rédaction d'un premier article intitulé "Multilayer

virtualized systems analysis with kernel tracing", présenté à la conférence ICI 2016 (The 3rd International Symposium on Intercloud and IoT). C'est sur cet article qu'est basé l'article du chapitre 4, en le généralisant aux VMs imbriquées à deux niveaux et aux conteneurs.

CHAPITRE 4

ARTICLE 1 : FINE-GRAINED MULTILAYER VIRTUALIZED SYSTEMS ANALYSIS

Authors

Cédric Biancheri
École Polytechnique de Montréal
cedric.biancheri@polymtl.ca

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

Keywords — Virtualized System ; KVM ; LXC ; Tracing ; LTTng

4.1 Abstract

With the consolidation of computer services in large cloud-based data centers, almost all applications and even application development execute in virtualized systems (VS's), sometimes nested. Whether it is inside a container, a virtual machine (VM) running on a physical host, or in a nested virtual machine, every process eventually runs on a physical CPU. Consequently, multiple virtualized systems might unknowingly compete with each other for physical resources. In this paper we study the interactions between all the VS's running on a physical machine. We introduce an analysis based on kernel tracing that erases the bounds between VS's and their host, to display a multilayer system as a single layer. As a result, it becomes possible to know exactly which process is currently running on a physical CPU, even if it is launched inside multiple layers of containers, themselves enclosed into two layers of VMs.

To use this analysis, we developed in Trace Compass a view that displays a time line for each host CPU, showing across time which process is running. Moreover, the full hierarchy of the VS's is retrieved from the analysis and is displayed in the view. By using a system of dynamic and permanent filters, we added the possibility to highlight in this view either traced VMs, virtual CPUs, specific processes and containers. This last feature, combined with

our view, allows to thoroughly apprehend the execution flow on the physical host, although it may involve multiple nested virtualized systems.

4.2 Introduction

Among the advantages of cloud environments we can cite their flexibility, their lower cost of maintenance, and the possibility to easily create virtual test environments. Those are some of the reasons explaining why they are widely used in industry. However, using this technology also brings its share of challenges in terms of debugging and detecting performance failures. Indeed, it can be more straightforward, when using the right tools, to detect performance anomalies while working with a simple layer of virtualization. For instance, if we have information about all the processes running on a machine through time, it is then possible to know for a specific thread which processes interrupted it. Because virtual machines (VM) are running in a layer independent of their host, it becomes more tedious to detect direct and indirect interactions between tasks happening inside a VM, on the host, inside a container, or even on nested or parallel VMs.

In this study, we focus on a way to analyze information, coming from a host, multiple VMs and linux containers (LXC) [45], as if all the execution was only happening on the host. The main objective is to erase as much as possible the boundaries between a host and the different virtual environments, to help a user visualize in a clearer way how the processes are interacting with each other.

To achieve this, we use kernel tracing on both the host and VMs, synchronize those traces, aggregate them into a unique structure and finally display the structure inside a view showing the different layers of the virtual environment during the tracing period. Considering the set of recorded traces as a whole system is the core concept of our fused virtualized systems (FVS) analysis presented here.

This paper is structured as follow : Section 2 exposes some related work about performance anomalies related to virtual environments. Section 3 explains in more details the multiple steps of the FVS analysis, including the single layered VMs (SLVMs), nested VMs (NVMs) and containers detection strategies. The same section introduces the view created to visualize the whole system. Section 4 presents some use cases for the FVS analysis and view. Section 5 concludes this paper.

4.3 Related Work

Dean et al. [46] created an online performance bug inference tool for production cloud computing. To accomplish this, they created an offline function signature extraction using closed frequent system call episodes. The advantage of their method is that the signature extraction can be done outside the production environment, without running a workload that usually triggers a performance default. By using their tool, they can identify a deficient function out of thousands of functions. However, their work is not adapted to performance anomalies involving multiple virtual machines.

In their work, Shao et al. [29] proposed a scheduling analyzer for the Xen Virtual Machine Monitor [17]. The analyzer uses a trace provided by Xen to reconstruct the scheduling history of each virtual CPU. By doing so, it is possible to retrieve interesting metrics like the block-to-wakeup time. However, this approach is limited to Xen and not directly applicable to other hypervisors. Furthermore, a trace produced by Xen is not sufficient to identify a process inside a VM that creates a perturbation across the VMs.

To gain in generality and not rely too much on hypervisors and application code, some work was initiated with the intention to detect performance anomalies across virtual machines by using kernel tracing.

With PerfCompass [47], Dean et al. used kernel tracing on virtual machines and created an online system call trace analysis, able to extract fault features from the trace. The advantage of their work is that it only needs to trace the virtual machine's system calls and not the host. Consequently, their solution has a low overhead impact and is able to distinguish between external and internal faults. However, it is not possible to see the direct interactions of the VM with neither the host nor the other VMs and the containers.

Another work proposed by Gebai et al. [28] focused more on the interactions between several machines. The authors proposed at first an analysis and a view showing, for each virtual CPU, when it is preempted. They also created a way to recover the execution flow of a specific process by crossing virtual machine boundaries to see which processes preempted it.

Their work is similar to ours but differs on multiple points. For instance, in their work, the Virtual Machine view displays one row for each virtual CPU. This number can easily grow if numerous VMs are traced. Consequently, the readability of the view can be altered. Additionally, by doing so, information about physical CPUs is lost. It is therefore impossible to track a VM, a virtual CPU or a process on the host. Finally, their work is dedicated to the analysis of single layered VMs, unlike our work that focuses also on nested VMs and

containers.

To our knowledge, no previous work tried to retrieve information about containers from a kernel trace. Other projects, like Docker [30], give access to runtime metrics such as CPU and memory usage, memory limit, and network IO metrics, exposed by the control groups [48] used by LXC. No previous work tries to represent the full execution of a multilayered system as if everything was happening on the host. Nonetheless, in reality, every process, even in nested VMs, eventually runs on a physical CPU of the host. Our contribution is to fulfill this gap.

4.4 Fused Virtualized Systems Analysis

A multilayered architecture is often the chosen strategy regarding the development of a software architecture. Each layer is dedicated to a specific role, independently of other layers, and is hosted by a tier, or a physical layout, that can contain multiple layers at once.

In this paper, we focus on a tier, or physical machine, hosting multiple layers of virtualized systems (VS) also called virtualized execution environment. A virtualized system will be considered as a virtual machine or a container. Figure 4.1 shows how the different layers can be organized in practical cases. Without using multilayers of virtual environments, the system is reduced to a single layer which is the host, also called the physical machine. This layer will be called L_0. Virtual machines adding a layer above the host will be labeled as L_1 VMs, and recursively, any VM above a L_n VM will be a L_{n+1} VM. Containers will not be labeled but will be associated to the machine directly hosting them. Containers can be running directly in L_0 but, for security reasons [49], they are most often used within virtual machines.

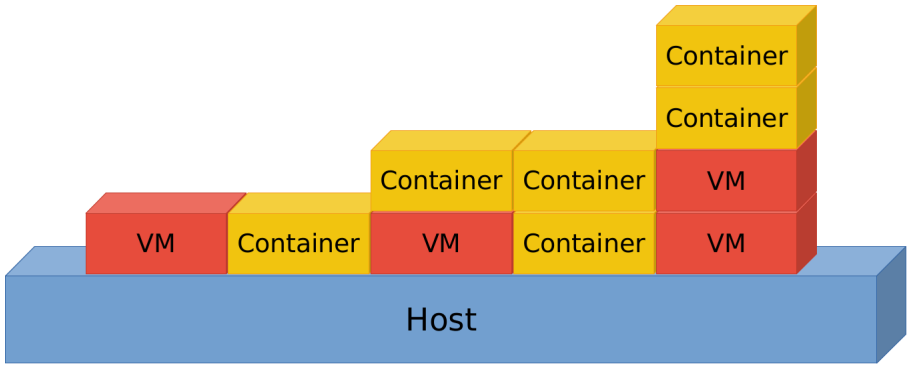

Figure 4.1 Examples of different configurations of layers of execution environment.

The idea we introduce here is to erase the bounds between L_0, its VMs in L_1 and L_2 and every container, to simplify the analysis and the understanding of complex multilayer architectures. Some methods for detecting performance degradations already exist for single-layer architectures. To reuse some of these techniques on multilayer architectures, one might remodel such systems as if all the activity was involving only one layer.

4.4.1 Architecture

The architecture of this work is described as follows : first we need to trace the host and the virtual machines, then because of clock drift [50] we have to synchronize those traces. After this phase, a data analyzer fuses all the data available from the different traces to put them in a data model. Finally, we need to provide an efficient tool to visualize the model that will allow the user to distinguish easily the different layers and their interactions. Those steps are summarized in Figure 4.2.

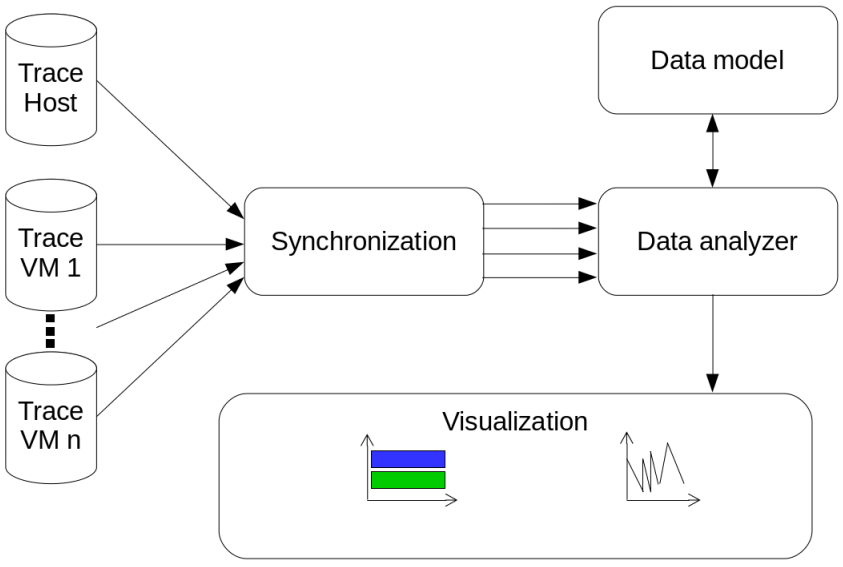

Figure 4.2 Architecture of the fused virtual machines analysis.

A trace consists of a chronologically ordered list of events characterized by a name, a time stamp and a payload. The name is used to identify the type of the event, the payload provides information relative to the event and the time stamp will specify the time when the event occurred.

In this study, we use the Linux Trace Toolkit Next Generation (LTTng) [4] to trace the

machine kernels. This low impact tracing framework suits our needs, although other tracing methods can also be adopted. By tracing the kernel, there is no requirement to instrument applications. Therefore, even a program using proprietary code can be analyzed by tracing the kernel. However, some events from the hypervisors managing the VMs are needed for the efficiency of the fused analysis. The analysis needs to know when the hypervisor is letting a VM run its own code or when it is stopped. Since, in our study, we are using KVM [18], merged in the Linux kernel since version 2.6.20 [51], and because the required trace points already exist, there is no need for us to add further instrumention to the hypervisor. In our case, with KVM using Intel x86 virtualization extensions, VMX [52], the event indicating a return to a VM running mode will always be recorded on L_0 and will be generically called a VMEntry. The opposite event will be called a VMExit.

Synchronization is an essential part of the analysis. Since traces are generated on multiple machines by different instances of tracers, we have no guaranty that a time stamp for an event in a first trace will have any sense in the context of a second trace. Figure 4.3 shows that without synchronization two traces recorded at the same time may seem to be created at two different times. The right scheduling of events, even coming from different traces, is crucial because when fusing the traces of a VM with its host, the events of the VM will have to be handled exactly between the VMEntry and the VMExit of L_0, relative to this specific VM. An imperfect synchronization can be the vector of incoherent observations that would impede the fused analysis. Figure 4.4 shows the difference between an analysis done on two pairs of traces with respectively an accurate and inaccurate synchronizations. The inexact synchronization can lead to false conclusions. In this case, a process from the VM seems to continue using the processor while in reality the VM has been preempted by the host.

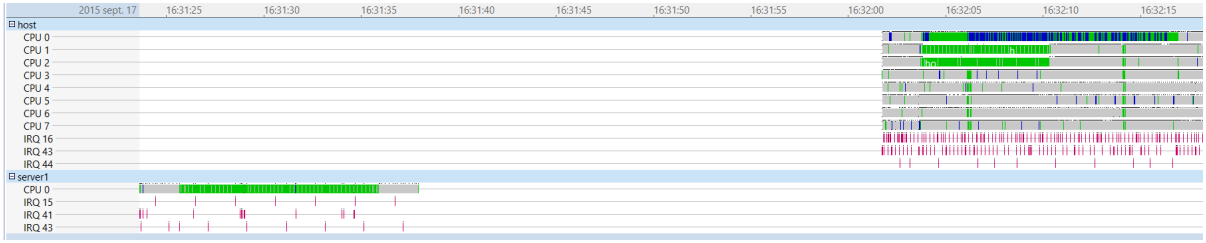

Figure 4.3 Traces visualization without synchronization.

Because VMs can be seen as nodes spread through a network, a trace synchronization method for distributed systems [37] can be adapted. As [28] we use hypercalls from the VMs to generate events on the host that will be related to the event recorded on the VM

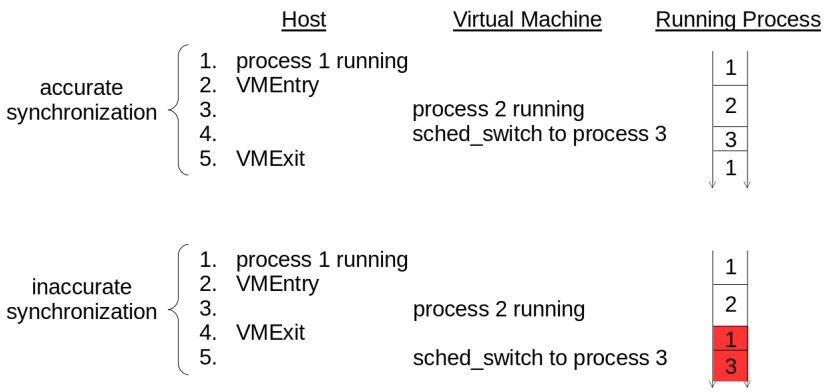

Figure 4.4 Wrong analysis due to inaccurate synchronization.

before triggering the hypercall. With a set of matching events, it is possible to use the fully incremental convex hull synchronization algorithm [53] to achieve trace synchronization. Because of clocks drift, a simple offset applied on the time stamps of a trace's events is not enough to synchronize the traces. To solve this issue, the fully incremental convex hull algorithm will generate two coefficients, a and b, for each VM trace while the host's trace is taken as time reference. Each event e_i will have its time stamp t_{e_i} transformed to t'_{e_i} with the formula :

$$t'_{e_i} = at_{e_i} + b$$

[28] used the hypercall only between L_0 and L_1. However, the method also applies between L_n and L_{n+1}, since an hypercall generated in L_{n+1} will necessarily be handled by L_n. In our case, synchronization events will be generated between L_0 and all its machines in L_1, and between machines of L_1 and their hosted machines. Consequently, a machine in L_2 will be synchronized with its host that will have previously been synchronized with L_0.

The purpose of the data analyzer is to extract from the synchronized traces all relevant data and to add them in a data model. Besides analyzing events specific to VMs and containers, our data analyzer should handle events generally related to the kernel activity. For this reason, the fused analysis is based on a preexisting kernel analysis used in Trace Compass [44], a trace analyzer and visualizer framework. Therefore, the fused analysis will by default handle events from the scheduler, the creation, destruction and waking up of processes, the modification of a thread's priority, and even the beginning and the end of system calls.

Unlike in a basic kernel analysis, the fused analysis will not consider each trace indepen-

dently but as a whole. Consequently, the core of our analysis is to recreate the full hierarchy of containers and VMs, and to consider events coming from VMs as if they were directly happening in L_0. As shown in Figure 4.5, for the simple case of SLVMs, the main objective is to construct one execution flow by fusing those occurring in L_0 and its VMs. The result is a unique structure encompassing all the execution layers at the same time, replacing what was seen as the hypervisor's execution, from the point of view of L_0, by what was really happening inside L_1 and L_2.

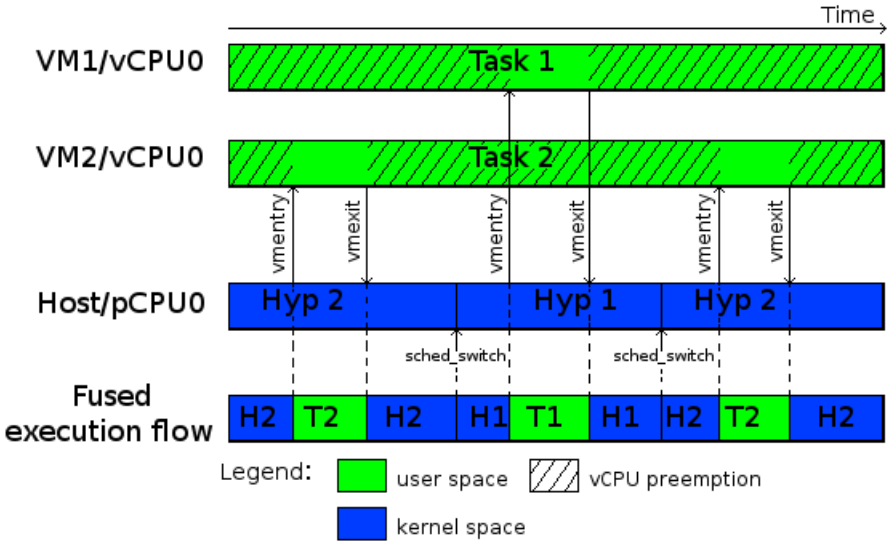

Figure 4.5 Construction of the fused execution flow.

KVM works in a way such that each vCPU of a VM is represented by a single thread on its host. Therefore, to complete the fused analysis, we need to map every VM's vCPU with its respective thread. This mapping is achieved by using the payloads of both synchronization and VMEntry events. On the one hand, a synchronization event recorded on the host contains the identification number of the VM, so we can match the thread generating the event with the machine. On the other hand, a VMEntry gives the ID of the vCPU going to run. This second information allows the association of the host thread with its corresponding vCPU.

4.4.2 Data model

The data analysis needs an adapted structure as data model. This structure needs to satisfy multiple criteria. A fast access to data is preferred to provide a more pleasant visualizer, so it should be efficiently accessible by a view to dynamically display information to users. The structure will also need to provide a way to store and organize the state of the whole system,

while keeping information relative to the different layers. For this reason, we need a design that can store information about diverse aspects of the system.

As seen in Figure 4.6, the structure contains information relating to the state of the different threads but also of the numerous CPUs, VMs and containers. Each CPU of L_0 will contain information concerning the layer that is currently using it, like the name of the VM running and with which thread and which virtual CPU. The Machine node will contain basic information about VMs and L_0, like the list of physical CPUs they have been using, their number of vCPUs or their list of containers. This node is fundamental since it is used to recreate the full hierarchy of the traced systems, in addition to the hierarchy of all the containers inside each machine.

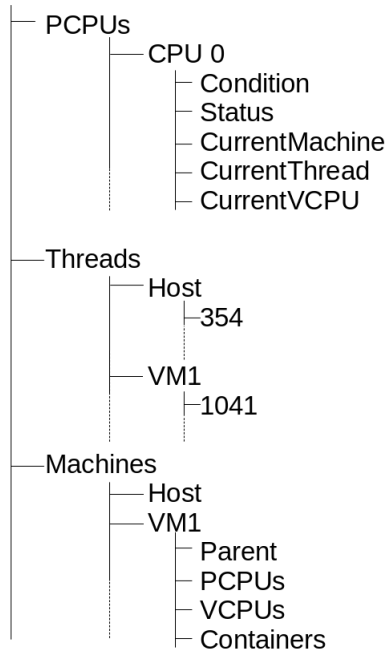

Figure 4.6 Structure of the data model.

Finally, the data model provides a time dimension aspect, since the state of each object attribute in the structure is relevant for a time interval. Those intervals introduce the need for a scalable model, able to record information valid from a few nanoseconds to the full trace duration.

In this study, we chose to work with a State History Tree (SHT) [12]. A SHT is a disk-based data structure designed to manage large streaming interval data. Furthermore, it provides an efficient way to retrieve, in logarithmic access time, intervals stored within this tree orga-

nization [54].

Algorithm 1 constructs the SHT by parsing the events in the traces. If the event was generated by the host, then the CPU that created the event is directly used to handle the event. However, if the event was generated by a virtual machine, we need to recursively find the CPU of the machine's parent harboring the virtual CPU that created the event, until the parent is L_0. Only then, the right pCPU is recovered and we can handle the event. This process is presented in Algorithm 2.

Algorithm 1 Handling multilayer kernel traces

Input : StateHistoryTree *s*, List <Event> *list*
1: **for** *each event in list* **do**
2: *machine* = Query the machine that generated *event* ;
3: **if** *machine is a VM* **then**
4: // translation between virtual and physical CPU
5: *cpu* = Query the pCPU currently running *machine*'s *cpu* ;
6: **else**
7: // the event happened in L_0
8: *cpu* = Query the CPU that generated *event* ;
9: **end if**
10: handleEvent(*s*, *event*, *cpu*) ;
11: **end for**

Algorithm 2 Retrieving the physical CPU

Input : Machine *machine*, Cpu *cpu*
Output : The physical CPU harboring *cpu*
1: **while** *machine is a VM* **do**
2: // *cpu* is a vCPU
3: *parent* = Query *machine*'s parent ;
4: *cpu* = Query *parent*'s CPU currently running *cpu* ;
5: *machine* = *parent* ;
6: **end while**
7: **return** *cpu*

The fundamental aspect of the construction of the SHT is the detection of the frontiers between the execution of the different machines and the containers. This detection is achieved by handling specific events and the application of multiple strategies.

Single Layered VMs Detection

In the case of SLVMs, the strategy is straightforward. The mapping is direct between the vCPUs of a VM in L_1 and its threads in L_0, a VM will be running its vCPU immediately after the recording of a VMEntry on its corresponding thread. Conversely, L_0 stops a vCPU immediately before the recording of a VMExit.

Algorithm 3 describes the handling of a VMEntry event for the construction of the SHT. In this case, we query the virtual CPU that is going to run on the physical CPU. Then, we restore the state of the virtual CPU in the SHT, while we save the state of the physical CPU. The exact opposite treatment is done for handling a VMExit event.

Algorithm 3 Handling vmentry event for Single Layered VMs

Input : StateHistoryTree *s*, Event *event*, Cpu *cpu*
1: **if** *event* == *vmentry* **then**
2: *vcpu* = Query the virtual CPU going to run on *cpu* ;
3: Save the state of *cpu* contained in *s* ;
4: Restore the state of *vcpu* in *s* ;
5: **end if**

Nested VMs Detection

For VMs in L_2, the previous strategy needs to be extended. Being a single-level virtualization architecture [22], the Intel x86 architecture has only a single hypervisor mode. Consequently, any VMEntry or VMExit happening at any layer higher or equal than L_1, is trapped to L_0. Figure 4.7 shows an example of the sequence of events and the hypervisors executions occurring on a pCPU when a VM in L_1 wants to let its guest execute its own code, and when L_2 is stopped by L_1. The dotted line represents the different hypervisors executing while the plain line shows when L_2 uses the physical CPU.

This architecture supersedes the previous strategy used for SLVMs. A VMEntry recorded in L_1 does not imply that a vCPU of a VM in L_2 is going to run immediately after. Likewise, L_2 does not yield a pCPU shortly before an occurrence of a VMExit in L_1, but when the hypervisor in L_0 is running, preceded by its own VMExit.

The challenge we overcome here is to distinguish which VMEntries in L_0 are meant for a VM in L_1 or L_2. Knowing that a VM of L_2 is stopped is straightforward, if the previous distinction is done. If a thread of L_0 resumes a vCPU of L_1 or L_2 with a VMEntry, then a VMExit from this same thread means that the vCPU was stopped.

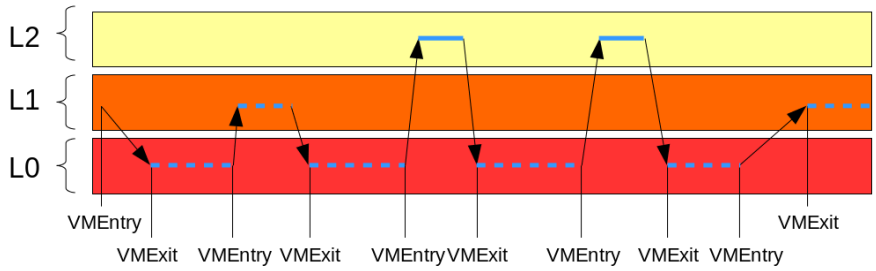

Figure 4.7 Entering and exiting L_2

We created two lists of threads in L_0. The waiting list and the ready list. If a thread is in the ready list, it means that the next VMEntry generated by this thread is meant to run a vCPU of a VM in L_2. The second part of Algorithm 4 shows that we retrieve the vCPU of L_2 going to run by querying it from the vCPU of L_1 associated to the thread. The pairing between the vCPUs of L_1 and L_2 is done in the first part of the algorithm, during the previous VMEntry recorded on L_1. It is also at this moment that the thread of L_0 is put in the waiting list.

Algorithm 4 Handling vmentry event for nested VMs

Input : StateHistoryTree *s*, Event *event*, Cpu *cpu*

1: **if** *event* == *vmentry* **then**
2: *vcpu* = Query the virtual CPU going to run on *cpu* ;
3: *machine* = Query the machine that generated *event* ;
4: **if** *machine is a VM* **then**
5: // the *vmentry* is from L_1 and *cpu* is a vCPU
6: Mark *cpu* as wanting to run *vcpu* ;
7: Mark L_0's thread running *cpu* as waiting ;
8: **return**
9: **end if**
10: // the *vmentry* is from L_0
11: *thread* = Query the thread running on *cpu* ;
12: **if** *thread* is ready for next layer **then**
13: // L_0's thread is ready to run L_2
14: // retrieve the real vCPU going to run
15: *vcpu* = Query the vCPU that *vcpu* wants to run ;
16: **end if**
17: Save the state of *cpu* contained in *s* ;
18: Restore the state of *vcpu* in *s* ;
19: **end if**

Algorithm 5 shows that the same principle is used for handling a VMExit in L_0. If the thread was ready, then we need again to query the vCPU of L_2 before modifying the SHT.

Algorithm 5 Handling vmexit event for nested VMs

```
Input : StateHistoryTree s, Event event, Cpu cpu
 1: if event == vmexit then
 2:     vcpu = Query the virtual CPU stopped ;
 3:     machine = Query the machine that generated event ;
 4:     if machine is a VM then
 5:         // vmexits are not relevant for L1
 6:         return
 7:     end if
 8:     // the vmexit is from L0
 9:     thread = Query the thread running on cpu ;
10:     if thread is ready for next layer then
11:         // L0's thread is stopping a vCPU in L2
12:         // retrieve the real vCPU stopped
13:         vcpu = Query the vCPU that vcpu was running ;
14:     end if
15:     Save the state of vcpu contained in s ;
16:     Restore the state of cpu in s ;
17: end if
```

When a thread of L_0 is put in the waiting list, it means that a vCPU of L_2 is going to be resumed. However, at this point, we don't know for sure which VMEntry will resume the vCPU. The kvm_mmu_get_page event solves this uncertainty by indicating that the next VMEntry of a waiting thread will be for L_2. Algorithm 6 shows the handling of this event and the shifting of the thread from the waiting list to the ready list.

Algorithm 6 Handling kvm_mmu_get_page event

```
Input : Event event, Cpu cpu
 1: if event == kvm_mmu_get_page then
 2:    machine = Query the machine that generated event ;
 3:    if machine is a VM then
 4:       // kvm_mmu_get_page is not relevant for VMs
 5:       return
 6:    end if
 7:    thread = Query the thread running on cpu ;
 8:    if thread is waiting then
 9:       Remove thread from the waiting list ;
10:       Mark thread as ready ;
11:    end if
12: end if
```

As seen in Figure 4.7, it is possible to have multiple entries and exits between L_0 and L_2 without going back to L_1. This means that a VMExit recorded on L_0 does not necessarily implies that the thread stopped being ready. In fact, the thread stops being ready when L_1 needs to handle the VMExit. To do so, L_0 must inject the VMExit into L_1 and this action is recorded by the kvm_nested_vmexit_inject event. Algorithm 7 shows that the handling of this event consists in removing the thread from the ready list.

Algorithm 7 Handling kvm_nested_vmexit_inject event

```
Input : Event event, Cpu cpu
 1: if event == kvm_nested_vmexit_inject then
 2:    machine = Query the machine that generated event ;
 3:    if machine is a VM then
 4:       // kvm_nested_vmexit_inject is not relevant for VMs
 5:       return
 6:    end if
 7:    thread = Query the thread running on cpu ;
 8:    Remove thread from the ready list ;
 9: end if
```

The process will repeat itself with the next occurrence of a VMEntry in L_1.

Containers Detection

The main difference between a container and a VM is that the container shares it's kernel with its host while a VM has its own. As a consequence, there is no need to trace a container since the kernel trace of the host will suffice. Furthermore, all the processes in containers are also processes of the host. Knowing if a container is currently running comes down to whether the current running process is from the said container or not.

The strategy we propose here is to handle specific events from the kernel traces to detect all the PID namespaces inside a machine. Then, we find out the virtual IDs of each thread (vTID) contained in a PID namespace.

A kernel trace generated with LTTng contains at least one state dump for the processes. A lttng_statedump_process_state event is created for each thread and any of its instances in PID namespaces. Furthermore, as seen in Figure 4.8, the payload of the event contains the vTID and the namespace ID (NSID) of the namespace containing the thread.

```
{ tid = 3887, vtid = 291, pid = 3881, vpid = 285, ppid = 3563, vppid = 1, ns_level = 1, ns_inum = 4026532199 }
{ tid = 3887, vtid = 3887, pid = 3881, vpid = 3881, ppid = 3563, vppid = 3563, ns_level = 0, ns_inum = 4026531836 }
{ tid = 3888, vtid = 292, pid = 3881, vpid = 285, ppid = 3563, vppid = 1, ns_level = 1, ns_inum = 4026532199 }
{ tid = 3888, vtid = 3888, pid = 3881, vpid = 3881, ppid = 3563, vppid = 3563, ns_level = 0, ns_inum = 4026531836 }
{ tid = 3893, vtid = 297, pid = 3893, vpid = 297, ppid = 3563, vppid = 1, ns_level = 1, ns_inum = 4026532199 }
{ tid = 3893, vtid = 3893, pid = 3893, vpid = 3893, ppid = 3563, vppid = 3563, ns_level = 0, ns_inum = 4026531836 }
```

Figure 4.8 Payload of lttng_statedump_process_state events.

Figure 4.9 shows how this information is added to the SHT. The full hierarchy of NSIDs and vTIDs is stored inside the thread's node to be retrieved later for the view. Moreover, each NSID and their contained threads are stored under it's host node. This allows to quickly know in which namespaces a thread is contained and, reciprocally, to known which threads belong to a namespace.

The analysis also needs to handle the process fork events to detect the creation of a new namespace or a new thread inside a namespace. In LTTng, the payload of this event provides the list of vTIDs of the new thread, besides of the NSID of the namespace containing it. Because the new thread's parent process was already handled by a previous process fork or a state dump, the payload combined with the SHT contains enough information to identify all the name spaces and vTIDs of a new thread.

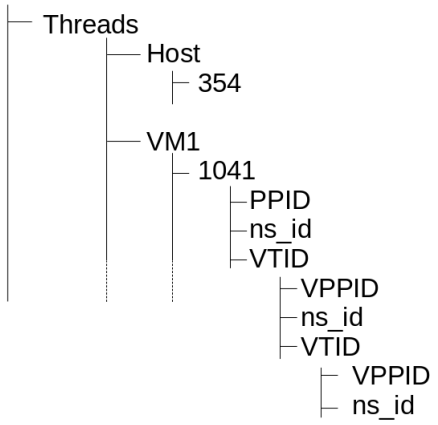

Figure 4.9 Virtual TIDs hierarchy in the SHT.

4.4.3 Visualization

After the fused analysis phase, we obtain a structure containing state information about threads, physical CPUs, virtual CPUs, VMs and containers through the traces duration. Our intention at this step is to create a view made especially for kernel analysis and able to manipulate all the information about the multiple layers contained inside our SHT. The objective is also to allow the user to see the complete hierarchy of virtualized systems. This view is called the Fused Virtualized Systems (FVS) view.

This view shows at first a machine's entry representing L_0. Each machine's entry of the FVS view can have at most three nodes. A PCPUs node, displaying the physical CPUs used by the machine, a Virtual Machine node, containing an entry for each of the machine's VM, and a Containers node, displaying one entry for each container. Because VMs are considered as machines, their nodes can contain the three previously mentioned nodes. However, a container will at most contain the PCPUs and Containers nodes. Even if it is possible to launch a VM from a container, we decided to regroup the VMs only under their host's node.

Figure 4.10 is a high level representation of a multilayered virtualized system. When traced and visualized in the FVS view, the hierarchy can directly be observed, as seen in Figure 4.11.

The PCPUs entries will display the state of each physical CPU during a tracing session. This state can either be idle, running in user space, or running in kernel space. Those states are respectively represented in gray, green and blue. However, there is technically no restriction on the number of CPU states, if an extension of the view is needed.

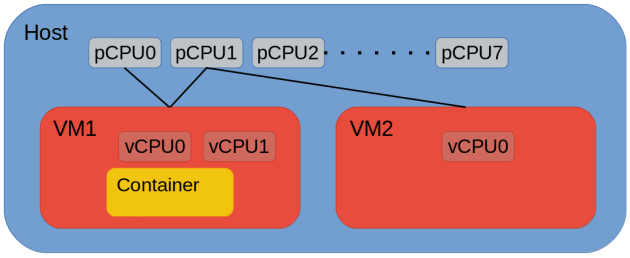

Figure 4.10 High level representation of a multilayered virtualized system.

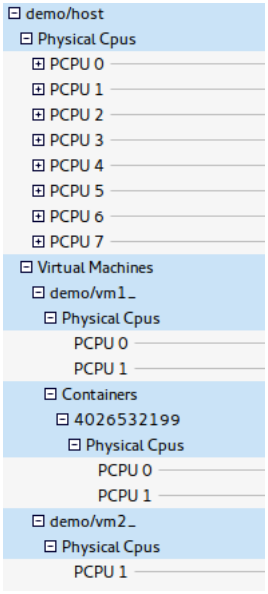

Figure 4.11 Reconstruction of the full hierarchy in the FVS view.

The Resources view is a time graph view in Trace Compass that is also used to analyze a kernel trace. It normally manages different traces separately and doesn't take into account the multiple layers of virtual execution. Figure 4.12 shows the difference between the FVS view and the Resources view displaying respectively a fused analysis and a kernel analysis coming from the same set of traces.

In this set, servers 1, 2 and 3 are VMs running on the host. All VMs are trying to take some CPU resources. As should be, the FVS view shows all the traces as a whole, instead of creating separate displays as seen in the Resources view. The first advantage of this configuration is that we only need to display the physical CPUs rows instead of one row for each CPU, physical or virtual. With this structure, we gain in visibility. The information from multiple

Figure 4.12 Comparison between FVS view and Resources view.

layers is condensed within the rows of the physical CPUs.

To display information about virtual CPUs, VMs and containers, the FVS view asks the data analyzer to extract some information from the SHT. Consequently, for a given time stamp, it is possible to know which process was running on a physical CPU, and on which virtual CPU and VM or container it was running, if the process was not directly executed on the host. Figure 4.13 shows the displayed tooltip when the cursor is placed on a PCPU entry. These are part of the information used to populate the entry.

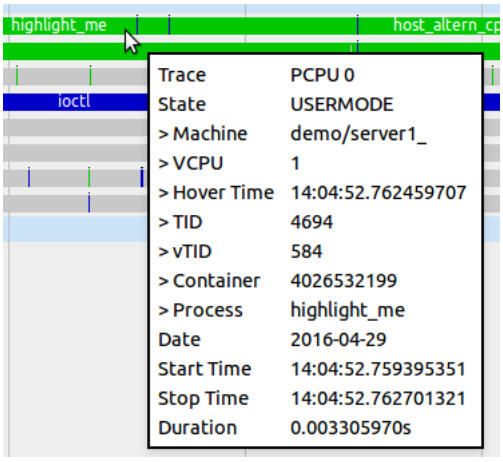

Figure 4.13 Tooltip displayed to give more information regarding a PCPU.

We noticed that, in the Resources view, the information is often too condensed. For instance, if several processes are using the CPUs, it can become tedious to distinguish them. Therefore, this situation is worse in the FVS view, because more layers come into play. For this reason, we developed a new filter system in Trace Compass that allows developers of time graph views to highlight any part of their view, depending on information contained in their data model.

Using this filter, it is possible to highlight one or more physical or virtual machines, containers, some physical or virtual CPUs, and some specifically selected processes. In particular, this filter will display what the user doesn't want to see, as if it was covered with a semi opaque white band. Selected areas will appear highlighted by comparison. Consequently, it is possible to see the execution of a specific machine, container, CPU or process directly in that view.

Figure 4.14 shows the real execution location of a virtual machine on its host. With this filter, we can distinctively see when the CPU was used by another machine, instead of the highlighted one.

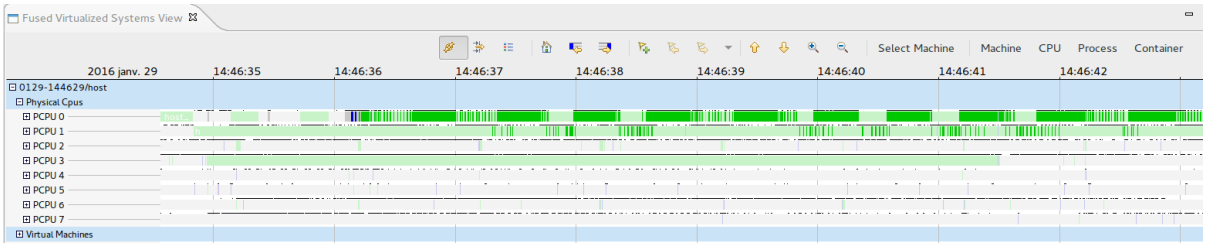

Figure 4.14 VM server1 real execution on the host.

In the FVS view, the states in the PCPUs entries of a virtualized system are a subset of the states visible in the PCPUs entries of the VS's parent. Only the physical host PCPUs display the full state history. The other entries can be considered as permanent filters dedicated to display only a VS and its virtualized subsystems. Figure 4.15 shows a magnified part of Figure 4.11 with all PCPUs nodes expanded. We can see that their sum equals the physical PCPUs entries.

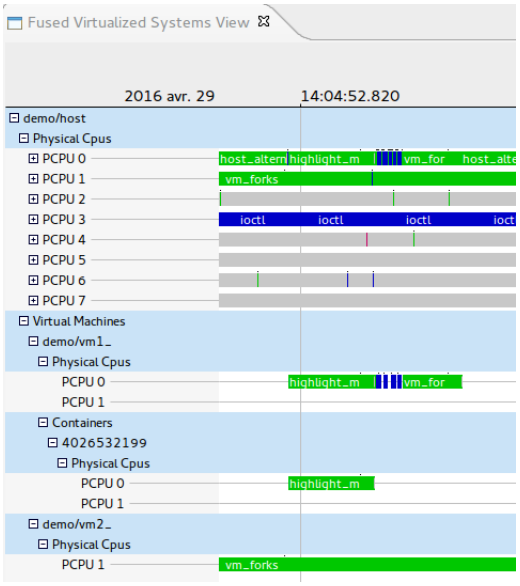

Figure 4.15 PCPUs entries of each virtualized system.

4.5 Use Cases and Evaluation

4.5.1 Use Cases

The concept of fusing kernel traces can have very interesting applications. In this section, we expose multiple use cases.

Our first use case is selecting a specific process, running in a container inside a virtual machine, in order to observe with the FVS view when and where the process was running.

Figure 4.16 shows that, from the point of view of the VM, the process vm_forks was running without interruption according to the Control Flow view. The Control Flow view is a view listing all the threads that were running during the tracing session, giving the state of those threads (running, waiting for CPU, blocked...). However, when we highlight the process in the FVS view, we clearly see that the selected process was preempted. If we magnify the view, we can even directly see which process from which machine is preempting our highlighted process, and when the process migrated to an other CPU.

Our next use case benefits from the fact that, by erasing the bounds between virtualized systems and the physical host, this analysis and view provide a tool to better understand the execution of an hypervisor. With the FVS view, it is possible to precisely see the interactions between the hypervisor and the host, depending on the instrumentation used.

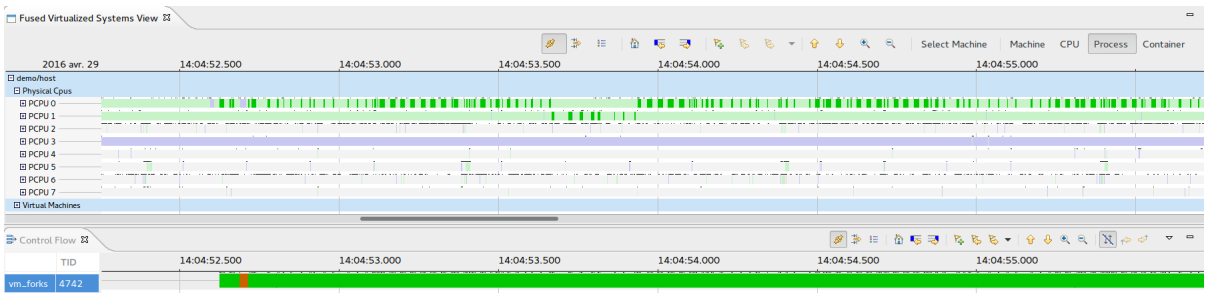

Figure 4.16 Highlighted process in the FVS view.

In our second use case, we propose to compare the time needed to wake up a sleeping process in L_1 and in L_2. In both L_1 and L_2, we created a process that sleeps for a short amount of time and then yields a PCPU. For both of them we examine the elapsed time between the wake up of the hypervisor in L_0 and the return to the VM's process. Figure 4.17 shows that resuming a VM in L_2 necessitates a lot of entries and exits between L_0 and L_1 due to trapped instructions. In our case, it took approximately 300 μs to wake up the process in L_2 while it took only 73 μs to wake up the one in L_1. This observed latency is a reason why deeper nested VMs suffer a higher virtualization overhead.

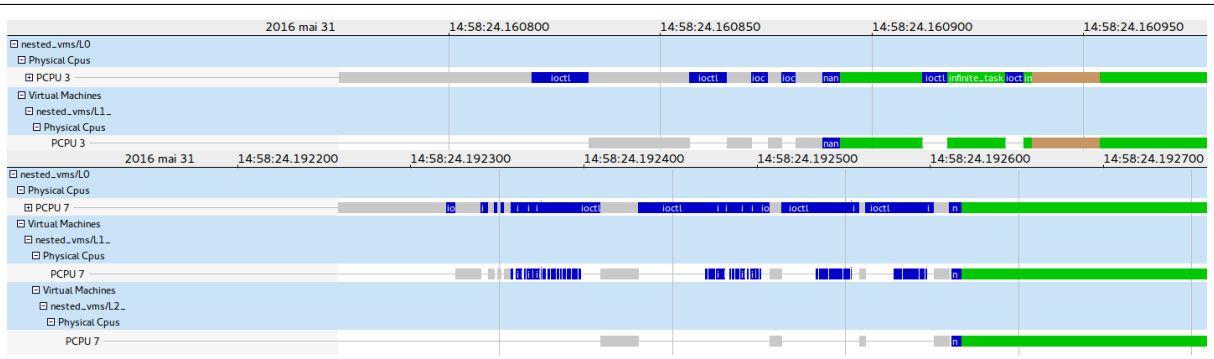

Figure 4.17 Process wake up time for L_1 and L_2.

Our third use case is observing how an interruption is handled inside a VM. Figure 4.18 shows what occurred during an I/O interruption happening in a VM running on physical CPU 1. We highlighted the execution of the VM to see when the hypervisor is involved. The hypervisor stopped the VM, meaning that the thread went out of guest mode, returned to kernel mode, then to user mode to handle the I/O interruption, then back to kernel mode

and finally let the VM run by switching back to guest mode. This behavior is completely consistent with what is expected in [18].

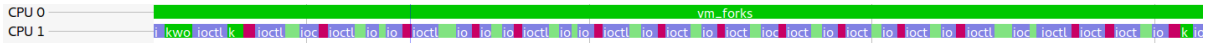

Figure 4.18 Handling of an ata_piix I/O interruption by the hypervisor on the physical CPU 1.

4.5.2 Evaluation

SHT's Generation Time

If we compare the time needed to complete a fused analysis for a set of traces and the one needed to complete a simple kernel analysis for the same set, we come to the conclusion that the simple kernel analysis is faster. Let T_i be the time needed to analyze trace i. Since the simple kernel analysis doesn't consider the set of traces as a whole but each trace independently, the analysis of the set can be done in parallel, each core dedicated to one trace. If we suppose that we have more cores than traces, then the elapsed time during the analysis will be $max_{1 \leq i \leq n} T_i$ where n is the number of traces.

If the set is considered as a whole, then it is difficult to process the traces in parallel. The elapsed time during the fused analysis will consequently be $\sum_{1 \leq i}^{n} T_i$.

Figure 4.19 shows experimentally the time needed for the fused analysis and a simple kernel analysis to build SHTs for different sizes of trace sets. We see that the build time for the fused analysis is directly related to the size of the trace set.

SHT's Size on Disk

To evaluate the space on disk necessary to realize the fused analysis, we compared the size of the SHT we created with the sum of the sizes of the SHTs created for each trace by the kernel analysis. Figure 4.20 shows that our SHT needs less space than the combined kernel analysis SHTs. However, we expected the sizes to be nearly equal since the fused analysis SHT can be seen as a combination of the kernel analysis SHT's. This gap is mainly explained

by the fact that the fused analysis starts to build the CPUs attributes of the SHT only when all the machine's roles have been determined.

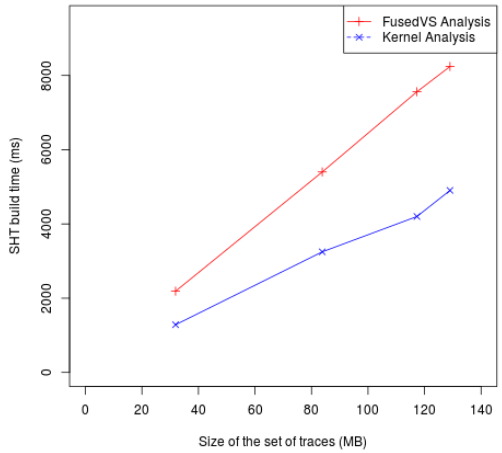

Figure 4.19 Comparison of construction time between FusedVS Analysis and Kernel Analysis.

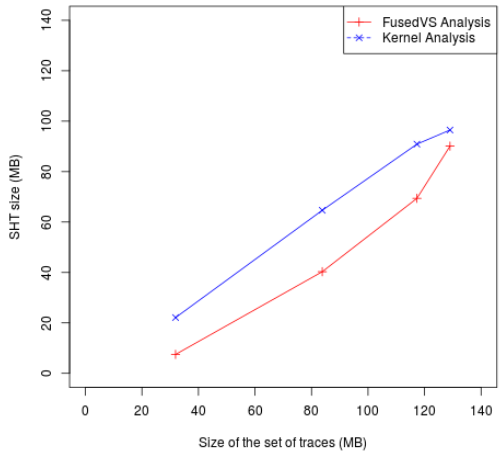

Figure 4.20 Comparison of the SHT's size between FusedVS Analysis and Kernel Analysis.

Those results were obtained with an Intel core i7-3770 and with 16GB of memory.

4.6 Conclusion and Future Work

In this paper, we presented a new concept of kernel trace analysis adapted to cloud computing and virtualized systems that can help for the monitoring and tuning of such systems and the development of those technologies. This concept is independent of the kernel tracer and hypervisor used. By creating a new view in Trace Compass, we showed that it was possible to display an overview of the full hierarchy of the virtualized systems running on a physical host, including VMs and containers. Finally, by adding a new dynamic filter feature to the FVS view, in addition to a permanent filter for any VS, we showed how it is possible to observe the real execution on the host of a virtual machine, one of its virtual CPUs, its processes and its containers.

In the future, we can expect the concept of the fused analysis to be reused and adapted for more specific utilization like the analysis of I/O or memory usage. We could also use the same principles to analyze more thoroughly systems using applications and programs in virtual execution environments, such as Java or Python. Finally, we can also extend our work to be able to visualize VMs' interactions between nodes to better understand the internal activity of cloud systems.

CHAPITRE 5

DISCUSSION GÉNÉRALE

5.1 Retour sur la visualisation

Les graphes temporels que l'on peut créer avec Trace Compass possèdent des fonctionnalités très utiles pour visualiser avec facilité des traces d'exécution. On peut créer une hiérarchie, agrandir la vue, ou bien filtrer certaines entrées pour n'afficher que celles d'intérêt. Cette dernière fonctionnalité est surtout pratique dans les vues affichant de nombreuses entrées, comme quand on visualise l'ensemble des processus du système. Cependant, il s'avère que ce filtre n'est pas suffisant dans la situation où l'on cherche à mettre en avant des transitions. Habituellement, les transitions dans ce type de vues sont représentées par des changement de couleurs. Dans la *Resources View*, on pourra observer la transition d'un CPU, de l'espace utilisateur vers l'espace noyau, par un changement de couleur du vert vers le bleu. Cependant, la transition d'un processus à un autre, tous les deux dans le même état, sera difficilement observable.

Le filtre présenté dans le chapitre 4 permet de palier à ce problème. Venant s'intercaler juste avant le tracé de chaque rectangle de couleur dans la vue, il permet de modifier dynamiquement la couleur avant de commencer à dessiner. Par ailleurs, le critère pour modifier la couleur est entièrement au choix du développeur. Originellement, le filtre avait été créé pour uniquement servir à mettre en avant l'exécution des machines virtuelles. Mais la simple modification des critères a permis de l'utiliser pour la mise en relief de conteneurs, des vCPUs, ainsi que des processus.

Initialement, la nouvelle couleur devait être la même que celle d'origine, avec l'ajout d'un coefficient alpha pour réguler sa transparence. Cependant, cette méthode s'avère coûteuse car le traçage d'une figure avec une couleur comprenant un coefficient alpha nécessite la lecture de chaque pixel du canevas avant d'écraser sa valeur. Ce procédé nuit grandement à la fluidité de la vue. C'est pour cette raison qu'une optimisation a été faite. Le fond du canevas étant toujours blanc, il est possible de contourner la lecture de chaque pixel à écraser en pré-calculant la nouvelle couleur sans lui attribuer un coefficient alpha. Cette optimisation permet de ne calculer qu'une seule fois la nouvelle couleur par rectangle à dessiner.

5.2 Détection de l'attribution des rôles

Dans l'analyse, le rôle de chaque machine est retrouvé grace à des évènements spécifiques aux systèmes hôtes et invités comme ceux de synchronisation. Cependant, il est primordial de pouvoir aussi déterminer le moment où chaque machine tracée a récupéré son rôle. L'analyse ne peut débuter sans cette information car elle est basée sur l'idée de retrouver le CPU physique qui a généré l'évènement, même s'il provient d'une couche de virtualisation. Pour traiter un évènement, il faut donc être capable, avant de récupérer le CPU physique, de déterminer s'il a été créé par une machine virtuelle et si l'hôte de cette machine est bien une machine physique. Tant que l'hôte est une machine virtuelle, il faut accéder à la machine parente. Dans le cas de machines virtuelles lancées directement à partir d'une machine physique, il ne peut y avoir plus qu'un rôle par machine. Par conséquent, si toutes les machines ont un rôle, cela implique que tous les rôles ont été trouvés et la fusion peut commencer. Cette solution ne s'applique pas aux VMs imbriquées car une machine peut être à la fois un hôte et un invité. La hiérarchie hôte/invité étant déterminée en même temps que la découverte d'une machine virtuelle, il est possible de prouver que les machines ont chacune leurs rôles en appliquant l'algorithme 8.

Algorithm 8 Détection de l'attribution des rôles

Input : List<Machine> *machines*
Output : Retourne vrai si les machines ont toutes leurs rôles

```
1: compteur = 0 ;
2: for machine in machines do
3:    if machine est Hôte et non Invité then
4:       compteur = Taille de la hiérarchie à partir de machine ;
5:       break ;
6:    end if
7: end for
8: return compteur == taille(machines) ;
```

L'algorithme consiste à parcourir toute la hiérarchie de machines à partir de l'une d'elle qui est un hôte mais pas un invité. Si le nombre de machines parcourues est égal au nombre de machines total, cela implique que toutes les machines ont leurs rôles. Sinon, il faut attendre la découverte d'un nouveau rôle pour à nouveau tester si l'attribution est complète.

5.3 Limitations de la solution proposée

Nous présentons dans cette section certaines limites de nos travaux. Le besoin de détecter l'intégralité des rôles des machines tracées avant de commencer la fusion peut être problématique. En effet, tant que la fusion ne débute pas, les évènements ne peuvent pas être traités. Ainsi, l'intervalle de temps représenté dans la vue aura toujours sa borne inférieure située après le lancement de la session de traçage de la dernière machine tracée. Par exemple, si la trace d'une VM s'étend sur les dix dernières secondes de la trace de son hôte, on ne pourra visualiser que ces dix dernières secondes.

La méthode de génération d'évènements utiles à la synchronisation utilise des hypercalls lors de sorties d'interruption logicielle. Ces hypercalls interrompent la VM et forcent l'hyperviseur à s'exécuter. Ces transitions sont coûteuses, surtout quand un hypercall est généré par une VM imbriquée. Le flot d'exécution alterne de nombreuses fois entre L_0 et L_1 avant d'être géré par L_1. Il en va de même pour le retour de L_1 à L_2.

La détection de conteneurs n'est pas complète. En réalité, ce n'est pas exactement les conteneurs qui sont retrouvés mais les espaces de noms des processus. Il est donc possible de voir apparaître plus d'espaces de noms qu'il n'y a de conteneurs réellement dans un système, si certains d'entre eux ne sont liés à aucun conteneur.

Enfin, la méthode pour déterminer quel niveau de VM est actuellement en train d'utiliser un CPU physique n'est adaptée qu'à des systèmes avec un ou deux niveaux de virtualisation. Une autre stratégie serait à développer pour pouvoir administrer plus de couches.

CHAPITRE 6

CONCLUSION ET RECOMMANDATIONS

6.1 Synthèse des travaux

Par l'intermédiaire de ces travaux, nous avons présenté un nouveau concept d'analyse de traces noyau pouvant aider le développement des technologies de virtualisation utilisées en infonuagique. Nous avons créé une analyse et une vue dans le logiciel Trace Compass permettant d'afficher d'une part l'intégralité de la hiérarchie des systèmes virtualisés tracés, et d'autre part leur exécution sur les processeurs physiques de l'hôte. Cette analyse, dite de fusion de traces, considère l'ensemble des traces produites comme un tout et se sert de différentes stratégies pour déterminer sur quel processeur de l'hôte une machine virtuelle est en cours d'exécution. L'arbre d'attributs, résultat de l'analyse, décrit l'intégralité des systèmes tracés comme si l'entièreté des flots d'exécution s'était déroulée sur un seul niveau. Dans le but de rendre notre analyse compatible avec des architectures incluant des machines virtuelles imbriquées, nous avons également étendu le modèle de machines virtuelles de Trace Compass pour le rendre compatible avec des systèmes exploitant deux niveaux de virtualisation.

Un système de filtrage performant a aussi été développé. Celui-ci permet aux utilisateurs de choisir quels flots d'exécution ils veulent observer. Grâce à celui-ci, il est possible de mettre en relief les différentes exécutions, comme celle d'une machine virtuelle, un CPU virtuel, un conteneur, ou un ensemble de processus. Cette représentation permet de mettre en avant des transitions qui étaient invisibles jusque là. Cette méthode de filtrage n'étant pas exclusive aux systèmes virtualisés, il est envisageable de facilement la réutiliser pour tout type de transitions.

En résumé, ce projet a été l'occasion de créer une analyse et une vue transformant une trace de bas niveau d'un système virtualisé multiniveaux, en une représentation graphique où tous les éléments de chaque niveau de virtualisation sont facilement observables sur leur hôte physique.

6.2 Améliorations futures

Plusieurs améliorations sont envisageables pour ce projet. Il serait pratique de remédier au problème de détection des rôles car une grosse partie de la trace peut être perdue. Pour cela,

deux options sont envisageables. Si l'on veut préserver l'automatisation de l'analyse, alors il faut retourner au début de la trace une fois que les rôles sont établis. Sinon, il est possible de demander à l'utilisateur de manuellement donner un rôle à chaque machine. En effet, on peut supposer que l'utilisateur est l'administrateur de son parc et qu'il connaît l'origine de chaque trace.

L'idée du fusionner des traces est encore sous-exploitée. Nous avons fourni ici la base servant à rétablir l'intégralité des exécutions avec les CPUs physiques les ayant réellement hébergées. Cependant, l'analyse n'offre pas de statistiques sur les différents systèmes virtualisés. On peut donc envisager des extensions de notre analyse permettant de faire ressortir des informations plus poussées. On pourrait donner, comme exemple d'information, le nombre de processus préemptés d'un conteneur lancé à partir d'une machine virtuelle durant un intervalle de temps, ainsi que la liste des processus responsables de ces préemptions.

La vue que nous avons créée pourrait aussi avoir d'autres utilisations. Par exemple, le filtre développé étant adaptable à n'importe quel critère, on pourrait imaginer la possibilité de mettre en relief des transitions basées sur des évènements que nous n'avons pas traités. Si un utilisateur active des points de trace pour signaler chaque entrée et sortie de fonction, alors rien ne l'empêchera d'utiliser notre filtre pour visualiser tous les instants où un processus choisi sera en train d'exécuter ces fonctions.

Enfin, on peut aussi imaginer la possibilité d'utiliser le concept de fusion à plus grande échelle, c'est à dire à travers les nœuds d'un système d'infonuagique. Au même titre que l'on peut actuellement voir les CPUs virtuels migrer d'un CPU physique à un autre, il serait possible d'observer une machine virtuelle être migrée d'un nœud physique vers un autre.

RÉFÉRENCES

[1] An introduction to kprobes. `http://lwn.net/Articles/132196/`. Accessed : 2016-07-09.

[2] Uprobes in 3.5. `https://lwn.net/Articles/499190/`. Accessed : 2016-07-09.

[3] Using the trace_event macro. `http://lwn.net/Articles/379903/`. Accessed : 2016-07-09.

[4] Mathieu Desnoyers and Michel R Dagenais. The lttng tracer : A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium)*, volume 2006, pages 209–224. Citeseer, 2006.

[5] ftrace - function tracer. `https://www.kernel.org/doc/Documentation/trace/ftrace.txt`. Accessed : 2016-07-09.

[6] trace-cmd : A front-end for ftrace. `http://lwn.net/Articles/410200/`. Accessed : 2016-07-09.

[7] Kernelshark. `http://people.redhat.com/srostedt/kernelshark/HTML/`. Accessed : 2016-07-09.

[8] perf examples. `http://www.brendangregg.com/perf.html`. Accessed : 2016-07-09.

[9] Lttng-ust vs systemtap userspace tracing benchmarks. `https://lists.lttng.org/pipermail/lttng-dev/2011-February/003949.html`. Accessed : 2016-07-09.

[10] Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, and Kenji Yoshihira. Uscope : A scalable unified tracer from kernel to user space. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–8. IEEE, 2014.

[11] I Buch and R Park. Improve debugging and performance tuning with etw. *MSDN Magazine,[Online],[Accessed : 01.01. 2012], Avaliable from : http ://msdn. microsoft. com/en-us/magazine/cc163437. aspx*, 2007.

[12] A Montplaisir-Gonçalves, N Ezzati-Jivan, Florian Wininger, and Michel R Dagenais. State history tree : an incremental disk-based data structure for very large interval data. In *Social Computing (SocialCom), 2013 International Conference on*, pages 716–724. IEEE, 2013.

[13] Antonin Guttman. *R-trees : a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[14] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification : Java SE 8 Edition*. Pearson Education, 2014.

[15] Anthony Velte and Toby Velte. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[16] Al Muller and Seburn Wilson. Virtualization with vmware esx server. 2005.

[17] Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Ian Pratt, Andrew Warfield, Paul Barham, and Rolf Neugebauer. Xen and the art of virtualization. 2003.

[18] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm : the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[19] VM Oracle. Virtualbox user manual, 2011.

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

[21] Advanced Micro Devices, *AMD64 Architecture Programmer's Manual Volume 2 : System Programming*, ch. 15. No. 24593, October 201.

[22] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project : Design and implementation of nested virtualization. In *OSDI*, volume 10, pages 423–436, 2010.

[23] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 26–35. ACM, 2008.

[24] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. Openstack : toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3), 2012.

[25] Nasa. `https://www.nasa.gov/`. Accessed : 2016-07-09.

[26] Rackspace : Managed dedicated & cloud computing services. `https://www.rackspace.com/`. Accessed : 2016-07-09.

[27] Tom White. *Hadoop : The definitive guide*. " O'Reilly Media, Inc.", 2012.

[28] Mohamad Gebai, Francis Giraldeau, and Michel R Dagenais. Fine-grained preemption analysis for latency investigation across virtual machines. *Journal of Cloud Computing*, 3(1) :1, 2014.

[29] Zhiyuan Shao, Ligang He, Zhiqiang Lu, and Hai Jin. Vsa : an offline scheduling analyzer for xen virtual machine monitor. *Future Generation Computer Systems*, 29(8) :2067–2076, 2013.

[30] Dirk Merkel. Docker : lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239) :2, 2014.

[31] Peter Fingar. *Dot cloud : the 21st century business platform built on cloud computing*. Meghan-Kiffer Press, 2009.

[32] The go programming language. `https://golang.org/`. Accessed : 2016-07-09.

[33] Lxd. `https://linuxcontainers.org/lxd/`. Accessed : 2016-07-09.

[34] Canonical, the company behind ubuntu. `http://www.canonical.com/`. Accessed : 2016-07-09.

[35] OS family Unix-like and GNU Userland. Ubuntu (operating system).

[36] Mark Masse. *REST API design rulebook*. " O'Reilly Media, Inc.", 2011.

[37] Masoume Jabbarifar. *On line trace synchronization for large scale distributed systems*. PhD thesis, École Polytechnique de Montréal, 2013.

[38] Ben Shneiderman. The eyes have it : A task by data type taxonomy for information visualizations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336–343. IEEE, 1996.

[39] Damien Dosimont. *Agrégation spatiotemporelle pour la visualisation de traces d'exécution*. PhD thesis, Grenoble Université, 2015.

[40] Robin Lamarche-Perrin. *Analyse macroscopique des grands systèmes : émergence épistémique et agrégation spatio-temporelle*. PhD thesis, Université de Grenoble, 2013.

[41] Damien Dosimont, Youenn Corre, Lucas Mello Schnorr, Guillaume Huard, and Jean-Marc Vincent. Ocelotl : Large trace overviews based on multidimensional data aggregation. In *Tools for High Performance Computing 2014*, pages 137–160. Springer, 2015.

[42] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1) :79–86, 1951.

[43] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1) :3–55, 2001.

[44] Trace compass. `http://tracecompass.org/`. Accessed : 2016-07-04.

[45] S. J. Vaughan-nichols. New approach to virtualization is a lightweight. *Computer*, 39(11) :12–14, Nov 2006.

[46] Daniel J Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. Perfscope : Practical online server performance bug inference in production cloud computing infrastructures. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[47] Daniel J Dean, Hiep Nguyen, Peipei Wang, and Xiaohui Gu. Perfcompass : toward runtime performance anomaly fault localization for infrastructure-as-a-service clouds. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*, 2014.

[48] Process containers. `http://lwn.net/Articles/236038/`. Accessed : 2016-07-04.

[49] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization : a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[50] Hicham Marouani and Michel R Dagenais. Internal clock drift estimation in computer clusters. *Journal of Computer Systems, Networks, and Communications*, 2008 :9, 2008.

[51] Linux 2 6 20. `http://kernelnewbies.org/Linux_2_6_20`. Accessed : 2016-07-04.

[52] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5) :48–56, 2005.

[53] Benjamin Poirier, Robert Roy, and Michel Dagenais. Accurate offline synchronization of distributed traces using kernel-level events. *ACM SIGOPS Operating Systems Review*, 44(3) :75–87, 2010.

[54] Alexandre Montplaisir, Naser Ezzati-Jivan, Florian Wininger, and Michel Dagenais. Efficient model to query and visualize the system states extracted from trace data. In *International Conference on Runtime Verification*, pages 219–234. Springer, 2013.