UNIVERSITÉ DE MONTRÉAL

SURVEILLANCE ET ANALYSE DE MACHINE VIRTUELLE ASSISTÉE PAR L'HÔTE

ABDERRAHMANE BENBACHIR
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)
MAI 2018

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

SURVEILLANCE ET ANALYSE DE MACHINE VIRTUELLE ASSISTÉE PAR L'HÔTE

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

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

DÉDICACE

À tous mes amis du labos,
vous me manquerez...

REMERCIEMENTS

Je tiens d'abord à remercier à mon directeur de recherche, Professeur Michel Dagenais, son expertise et son mentorat m'ont été essentiels pour compléter ce travail.

J'aimerais remercier mes camarades du laboratoire DORSAL, leurs conseils, soutien et interactions ont permis d'améliorer le travail réalisé dans ce projet.

J'aimerais aussi remercier toute ma famille, particulièrement mes parents Souad ELHAJOUI et Youssef BENBACHIR, qui m'avaient toujours encouragé à poursuivre des études supérieures. Je remercie infiniment ma conjointe Abir Tabtoubi, avec sa patience et son accompagnement continus j'ai pu persister et atteindre mes objectifs, même lors des périodes difficiles au cours de la réalisation de ce projet.

Finalement, je tiens à exprimer ma gratitude pour le soutien financier offert par Ericsson, Ciena, Google, EfficiOS et le projet de recherche CRSNG/Prompt ainsi que leurs interactions utiles qui ont rendu cette étude possible.

RÉSUMÉ

L'arrivée des extensions de processeurs Intel VMX et AMD SVM ont rendu possible la virtualisation de la plateforme x86 en exécutant des systèmes invités non modifiés. Les technologies de virtualisation telles que KVM et Xen sont devenues de plus en plus populaires et sont largement adoptées par l'industrie et les groupes de recherche pour une variété d'applications. Les systèmes virtualisés bénéficient de l'isolation virtuelle offerte par la virtualisation, ce qui donne l'illusion du contrôle absolu sur les ressources de l'hôte. Cette isolation affecte la performance de toute application s'exécutant dans un environnement de machine virtuelle, principalement en raison du surcoût induit par les interactions avec l'hyperviseur et aussi les interactions avec d'autres machines virtuelles cohabitant sur la même machine. Cependant, il est possible de bénéficier de cette fonctionnalité pour investiguer des causes profondes de dégradation de performance pendant que le système passe par des phases critiques comme le démarrage et l'arrêt, ce qui est très difficile à surveiller dans un environnement non virtualisé. L'objectif de cette étude est de fournir une infrastructure de surveillance basée sur des techniques de paravirtualisation qui facilite la collaboration entre l'hôte et les invités et permet ainsi une détection précise des temps de latence.

Pour atteindre cet objectif, nous utilisons des canaux de communication, l'hypercall et la mémoire partagée, des techniques basées sur la paravirtualisation que nous avons développées dans le traceur Ftrace. Notre approche fonctionne à travers l'infrastructure de l'hyperviseur pour faciliter le partage des données des systèmes invités, sans recourir aux opérations d'E/S utilisant le réseau et le disque, car les deux ne sont pas disponibles pendant le démarrage ou l'arrêt de la VM. De plus, en utilisant ces opérations d'E/S, les machines virtuelles souffrent d'une baisse significative des performances. Enfin, nous avons développé une optimisation KVM afin de supprimer la multiplication de sortie et permettre à notre approche de surveiller efficacement les environnements imbriqués.

ABSTRACT

The introduction of hardware-assisted virtualization capabilities support, in both Intel VMX and AMD SVM processor extensions, made x86 virtualization possible while running unmodified OS guests. Virtualization technologies such as KVM and Xen have become increasingly popular, and are widely adopted by industry and researchers for a variety of applications. Virtualized systems benefit from the virtual isolation offered by virtualization, which gives the illusion of absolute control over the host resources. This isolation impact the performance of any application running in a virtual machine environment, mostly because of the overhead induced from interactions happening with the host hypervisor and other co-located virtual machines. However, it is possible to benefit from virtualization features to find the root cause of performance problems while a system is executing in critical phases like boot-up and shutdown. During these phases, very few communication channels are available (e.g. only serial ports) and it is very difficult to monitor the execution in a non-virtualized environment. The objective of this study is to provide a paravirtualization-based monitoring infrastructure which facilitates host and guest collaboration and enables accurate latency detection.

To accomplish this objective, we use hypercall and shared memory communication channels, a paravirtualization-based technique that we developed within the Ftrace tracer. Our approach relies on the hypervisor infrastructure to allow the guest trace data to be shared without relying on I/O operations from devices like the network and disk, because neither is available while a VM is booting up or shutting down. Moreover, when using I/O operations, VMs suffer from a significant performance drop. Finally, we developed a KVM optimization in order to remove exit multiplication and enable our approach to efficiently monitor nested environments.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

AMD	Advanced Micro Devices
BCC	BPF Compiler Collection
BPF	Berkeley Packet Filter
BT	Binary translation
CLI	Clear Interrupts
CPU	Central Processing Unit
CR3	Control Register 3
CTF	Common Trace Format
eBPF	Enhanced Berkeley Packet Filter
EPT	Extended Page Table
E/S	Entrées/Sorties
FreeBSD	Free Berkeley software distribution
HLT	Halt
ISA	Instruction Set Architecture
KGDB	Kernel GNU Project Debugger
KVM	Kernel-based Virtual Machine
QEMU	Quick Emulator
OS	Operating System
PCI	Peripheral Component Interconnect
SVM	Secure Virtual Machine
TCG	Tiny Code Generator
VMCS	Virtual Machine Control Strucutre
VM	Virtual Machine
VMM	Virtual Machine Monitor
VT	Virtualisation Technology
vCPU	Virtual CPU

CHAPITRE 1 INTRODUCTION

L'infonuagique (Cloud) permet aux fournisseurs d'allouer des ressources purement à la demande, et de varier la quantité de ressources pour répondre à la demande de charge de travail. Le potentiel de l'infonuagique réside dans son modèle « payer au fur et à mesure », c.-à-d. les utilisateurs ne paient que les ressources qu'ils sont effectivement utilisés et peuvent librement contrôler de manière flexible la capacité de ressources qui leur est attribuée à tout moment [4].
Cependant, un fait important dans ce processus dynamique est que, bien que les utilisateurs de Cloud puissent faire leurs demandes d'acquisition à tout moment, ce processus peut prendre un certain temps avant que les ressources acquises ne soient prêtes à utiliser. Les fournisseurs de Cloud ont besoin de temps pour trouver un endroit pour approvisionner une machine virtuelle (VM) dans leurs centres de données [37]. En fait, ce processus dynamique est inévitable pour toutes les VMs pour assurer l'élasticité du nuage. Une longue latence indésirable au cours de ce processus pourrait entraîner une dégradation de la réactivité de l'élasticité, qui nuira immédiatement à la performance de l'application.

Les systèmes virtualisés bénéficient de l'isolement absolu offert par la virtualisation, ce qui leur donne l'illusion d'avoir le contrôle absolu et exclusif sur les ressources matérielles. Les systèmes invités résidant sur la même machine hôte n'ont aucun accès au système hôte dans lequel ils s'exécutent. Ils sont continuellement en compétition pour l'accès aux ressources matérielles, et peuvent dans la plupart des cas interférer entre eux, conduisant à des latences réelles, mais invisibles (pour les invités) et facilement mesurables depuis l'hôte.

Le système hôte a pleinement accès aux ressources, mais n'a pas accès au flot d'exécution interne des systèmes virtuels. Afin d'offrir un meilleur service de partage de ressources aux systèmes virtuels, on utilise la technique de paravirtualisation.

1.1 Définitions et concepts de base

Pour mieux comprendre le travail réalisé dans le présent mémoire, on introduit dans cette section quelques concepts de base à étudier.

1.1.1 Introduction au traçage

Le traçage est une technique de journalisation sophistiquée permettant de récupérer l'information d'exécution (ou *runtime*) sans arrêter les processus. Le traçage se fait en enregistrant

l'exécution chronologique des évènements émis par les points de trace (ou *tracepoint*) insérés soit dans les applications ou au niveau du noyau du système. Dans le contexte du traçage, cette insertion est connue comme "instrumentation".

L'instrumentation peut être réalisée par deux approches. La première est l'instrumentation statique, cette dernière implique un changement préalable au code source. Quand l'insertion d'un point de trace est faite, la position et les arguments passés aux points de traces ne peuvent pas être changés. Cette caractéristique rend difficile la maintenance des points de trace durant le cycle de vie d'un logiciel. La deuxième approche est l'instrumentation dynamique, l'insertion des points de trace se fait dynamiquement durant l'exécution du logiciel, sauf que le coût associé à l'utilisation de cette approche n'est pas négligeable comparer à l'approche statique [18].

Les évènements récupérés durant le traçage sont des faits importants lors de l'analyse du comportement du système. Un évènement est ponctuel de par sa nature, il peut être ordonné chronologiquement par l'estampille de temps (*timestamp*). En plus, l'évènement rapporte d'autres informations (*payload*) reflétant l'état du système à un instant donné.

1.1.2 Machine virtuelle

Une machine virtuelle (ou VM) est un terme générique, souvent utilisé pour désigner un environnement isolé exécutant des flots d'instructions partiellement visibles au système hôte. Cette couche d'isolation offre une interface d'abstraction flexible offrant une forte portabilité et sécurité aux logiciels exécuté par la machine virtuelle.

Plusieurs technologies utilisent la machine virtuelle pour fin d'émulation. Cependant, le cadre de ce travail sera restreint au contexte infonuagique. Particulièrement, nous allons étudier les systèmes d'exploitation virtualisés (ou systèmes invités) qui sont exécutés dans une machine virtuelle. Pour offrir une telle isolation à une machine virtuelle, la machine hôte doit implémenter la virtualisation pour simuler le logiciel ainsi que le matériel. La simulation matérielle est fortement répandue dans l'industrie informatique, car elle permet de diminuer les coûts associés à l'achat de nouveau matériel informatique.

Toutefois, les coûts de performance associés à la virtualisation ne sont pas négligeables. La dégradation de performance mesurée peut varier en fonction de la nature de la charge de travail exécutée sur la machine virtuelle. Elle peut atteindre 20% comparée à l'exécution native sur le matériel. Ceci dit, des techniques de virtualisation avancées comme la paravirtualisation et la virtualisation assistée par matériel furent introduites pour optimiser le surcoût associé à la couche de virtualisation.

1.1.3 Paravirtualisation

La paravirtualisation est un mécanisme de communication entre l'hôte et ses invités ayant pour but d'améliorer la performance. L'efficacité de cette communication est réalisée grâce à la coopération du système d'exploitation invité avec l'hôte, au moyen d'une interface permettant au système invité d'indiquer son intention à l'hyperviseur.

Pour ce faire, les systèmes invités doivent être modifiés, soit en remplaçant les instructions privilégiées par une seule instruction *vmcall*, permettant une communication synchrone et directe avec l'hôte, soit en configurant une mémoire partagée avec l'hôte comme moyen d'échange de données de façon asynchrone. [38].

1.2 Éléments de la problématique

Les différents problèmes de performance dans un environnement virtuel sont principalement dus à la couche d'isolation imposée par l'hyperviseur. Les machines virtuelles ont l'illusion d'un accès exclusif au matériel physique, en raison de la couche de virtualisation. Ceci apporte des latences nouvelles liées au désengagement de ressources (partage du CPU) ou à l'émulation d'instructions privilégiées.

Pour être en mesure d'étudier ces problèmes dans un système aussi complexe, l'activation de la surveillance dans toutes les couches est l'approche triviale, normalement la plus utilisée. Toutefois, cette approche vient avec un coût, la plupart des outils de surveillance doivent consommer des données enregistrées d'une manière ou d'une autre, soit en les stockant sur le disque pour une analyse hors connexion, soit en utilisant la surveillance en direct pour les décharger par réseau pour l'analyse en ligne. Les deux approches introduisent des coûts liés aux opérations d'entrées-sorties (E/S), en particulier lorsque des données à haute fréquence sont générées dans des systèmes virtualisés.

Le démarrage est la quantité de temps qu'il faut pour démarrer une machine virtuelle jusqu'à ce qu'elle atteigne un état prêt. C'est un facteur important pour les stratégies d'allocation de VM, en particulier durant la phase de provisionnement pour répondre à la demande [45]. Les stratégies d'allocation automatisées demanderont aux nouvelles VM de répondre à la demande de charge. Dans cette situation, le temps de démarrage est critique, le système est en attente après le démarrage afin d'être pleinement opérationnel [13]. L'activation de la surveillance pendant le démarrage est difficile, bien que ce soit la seule façon pour examiner les latences de démarrage. De plus, les données générées durant le démarrage ne peuvent pas être stockées sur disque ou déchargées via le réseau, car le réseau et le stockage ne sont disponibles qu'à la fin de cette phase.

Les plantages du noyau peuvent également se produire durant le démarrage des machines virtuelles. L'étude de tels problèmes est une tâche non triviale. Les développeurs du noyau utilisent seulement la sortie de la console série pour afficher les messages d'erreur ou les valeurs des registres.

Les latences liées à la phase d'arrêt d'un système sont aussi très difficiles à déboguer. Ces latences peuvent ne pas être si importantes lorsqu'un système est en arrêt complet. Toutefois, dans le cas où le système doit redémarrer après une mise à jour majeure, toute latence importante devient immédiatement critique. Lorsque le système est en cours d'arrêt, les processus utilisateur sont libérés, les pilotes sont en cours d'arrêt, le réseau et le disque peuvent ne pas être disponibles. Les outils de surveillance actuels ne sont pas conçus pour

fonctionner dans cette phase.

Dans le chapitre 3, nous allons présenter comment nous avons utilisé la paravirtualisation pour concevoir des outils de communication, incorporés aux outils de traçage, pour investiguer efficacement de tels problèmes.

1.3 Objectifs de recherche

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

Est-il possible de développer une technique de paravirtualisation (collaboration hôte-invité), permettant d'investiguer les problèmes de baisse de performance dans les machines virtuelles ?

Cette technique peut-elle être utilisée pour investiguer les problèmes survenant lors de l'initialisation et de l'arrêt des systèmes invités ?

Nous avons identifié nos objectifs de recherche de la façon suivante :

1. Aider au débogage et à l'investigation des problèmes d'exécution des machines virtuelles (OS invité) en développant des techniques de paravirtualisation.
2. Aider à trouver les causes des latences reliées aux phases sensibles des machines virtuelles à l'aide de nouveaux algorithmes d'analyse de traces d'exécution.
3. Proposer une solution de fusion des évènements hôte et invités à faible coût.
4. Établir un protocole de communication efficace (uni/bidirectionnel) entre l'hôte et ses invités.

1.4 Plan du mémoire

Le structure du mémoire est organisée comme suit : le chapitre 2 présente une revue de littérature commençant par le traçage et les outils de visualisation, et poursuivant avec la virtualisation et les techniques de paravirtualisation adaptées au monitorage. Le chapitre 3 présente la méthodologie envisagée pour relever les défis annoncés précédemment. Le chapitre 4 réfère à l'article de journal "Hypertracing : Tracing Through Virtualization Layers". Le chapitre 5 présente une rétroaction sur l'article. Le chapitre 6 conclut ce travail en établissant les orientations futures.

CHAPITRE 2 REVUE DE LITTÉRATURE

2.1 Surveillance et monitorage

Les outils de trace dans Linux sont très nombreux et sont connus pour leur efficacité à collecter les données sans impacter le système observé. Tel qu'illustré à la figure 2.1, ces outils peuvent être classés dans deux catégories distinctes : Traceur et Agrégateur [23].

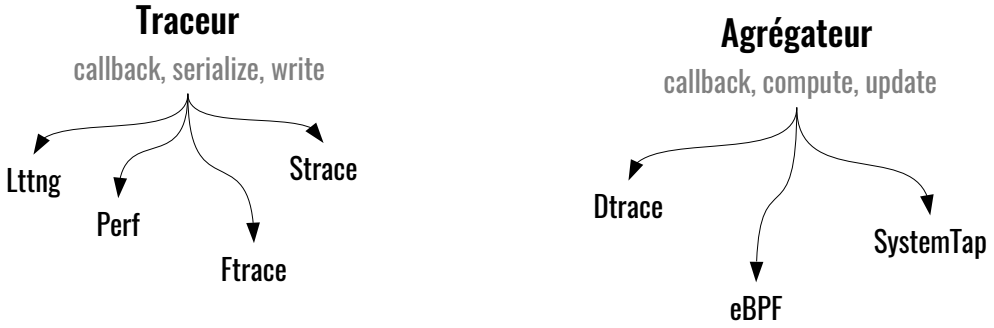

Figure 2.1 Outils de traçages et profilage

Un **traceur** est un outil dédié à collecter l'information la plus optimale possible. Toute information collectée est généralement enregistrée sur disque afin d'appliquer des analyses hors ligne (ou *offline analysis*). Contrairement aux traceurs, les outils comme Dtrace [41] ou SystemTap [22] sont considérés des agrégateurs. Un **agrégateur** agrège continuellement et en temps réel l'information collectée sous forme de métrique. Des résultats statistiques sont généralement présentés à la sortie sous forme d'histogrammes.

Dans cette section, nous allons étudier quelques outils de monitorage particuliers, qui sont fortement supportés et maintenus par la communauté open source.

2.1.1 Ftrace

Ftrace [53] est un traceur complètement interne au noyau Linux, originalement développé dans le projet PREEMPT_RT [1] puis fusionné en amont (ou *upstream*) dans la branche principale du noyau Linux en 2008. Ftrace peut tracer les évènements statiques en utilisant la macro TRACE_EVENT [55]. Il peut aussi placer des points de trace dynamiquement en utilisant l'infrastructure de *kprobes* [39], et peut en outre tracer les évènements d'entrée et sortie des fonctions, en utilisant l'appel mcount() injecté au début de chaque fonction lorsque l'option *-pg* est activée dans le compilateur *gcc* .

La manipulation de Ftrace est possible en utilisant l'interface *debugfs* sous forme de système de fichiers, monté par défaut sur le répertoire */sys/kernel/debug*. Un répertoire nommé *tracing* présente plusieurs fichiers de configuration pouvant être manipulée par les commandes *bash* "cat" et "echo".

```
$ cat available_events
$ echo 1 > events/sched/sched_switch/enable
$ echo 1 > tracing_on; sleep 1; echo 0 > tracing_on
$ cat trace
```

La manipulation directe de l'interface *debugfs* est très difficile, trace-cmd [54] a été développé spécialement pour ce cas, afin d'offrir une interface flexible ainsi qu'offrir un moyen d'enregistrement de la trace en format binaire. L'exemple suivant présente l'utilisation des commandes illustrées précédemment, mais cette fois-ci avec l'outil trace-cmd.

```
$ trace-cmd record -e sched_switch -o /tmp/trace.dat
$ trace-cmd report /tmp/trace.dat
cpus=4
    trace-cmd-24631 [003] 34367.582359: sched_switch:
                            trace-cmd:24631 [120] S ==> kworker/u16:1:23740 [120]
kworker/u16:1-23740 [003] 34367.582388: sched_switch:
                            kworker/u16:1:23740 [120] S ==> terminator:23293 [120]
    trace-cmd-24632 [002] 34367.582512: sched_switch:
                            trace-cmd:24632 [120] S ==> gmain:1714 [120]
...
```

Ftrace offre aussi une grande liste de profils de traçage (ou *modules*) prédéterminés qui peuvent être listés avec la commande suivante :

```
$ cat available_tracers
blk function_graph function hwlat irqsoff preemptoff wakeup nop
```

Chaque module est configurable par le fichier *trace_options*. Ces options sont implémentées en utilisant la macro TRACER_OPT pour définir le nom et le bit (*flag*) associés à chaque option, cette technique permet par la suite des manipulations flexibles et efficaces. Certains modules permettent la détection des latences souvent causées par différent scénarios comme : la désactivation des interruptions (module *irqsoff*), la désactivation de préemption (module *preemptoff*), des latences liées à l'ordonnancement (module *wakeup*) ou soit des latences matérielles (module *hwlat*).

Ftrace offre aussi une fonctionnalité très utile appelée les déclencheurs de traçage (ou *trace triggers*). Un déclencheur permet d'activer une ou plusieurs commandes quand un ou plusieurs évènements sont enclenchés, cette technique permet à l'utilisateur de réduire la quantité d'information collectée, de raffiner l'investigation d'un comportement particulier (Bug).

Ftrace offre deux types d'évènements déclencheurs. Le premier est le déclencheur de fonction (*function triggers*), qui permet d'invoquer des commandes lors d'un appel de fonction. Le deuxième est le déclencheur d'évènement (*trace event triggers*), qui permet aussi d'invoquer des commandes quand un évènement spécifique est atteint. Les deux déclencheurs offrent le filtrage conditionnel (*event filtering* et *function filtering*) en associant des expressions booléennes et des opérateurs relationnels (==, !=, <, <=, >, >=, &, *, ?).

```
$ echo 'command[:count] [if filter]' > /events/.../trigger
```

Les commandes supportées par Ftrace sont définies comme suit :

enable_event/disable_event : Ces deux commandes permettent l'activation ou désactivation des évènements de trace à chaque fois que l'évènement déclencheur est atteint [52].

L'opération d'activation et désactivation d'évènements est très coûteuse, car elle nécessite l'usage de la fonction stop_machine() afin d'arrêter tous les CPUs pendant que les changements effectués en cours d'exécution (*live patching*) sont propagés sur tout le système. Pour remédier à cela, Ftrace implémente une technique de désactivation douce (ou *soft disable*). Le mode doux active le point de trace puis le marque comme étant "désactivé" en utilisant le *flag* SOFT_DISABLE, le point de trace sera toujours appelé sans émettre d'évènements. Il restera dans ce mode tant qu'il y a un déclencheur en cours d'exécution [21].

Par exemple, le déclencheur suivant permet le traçage de l'évènement *kmalloc* lorsqu'un appel système d'écriture vient de commencer, le " :1" défini que l'activation du traçage s'effectuera quand l'évènement sera rencontré dès la première fois [52].

```
$ echo 'enable_event:kmem:kmalloc:1' > events/syscalls/sys_entry_write/trigger
```

Le déclencheur suivant permet la désactivation du traçage pour les évènements *kmalloc* lorsqu'un appel système d'écriture est terminé.

```
$ echo 'disable_event:kmem:kmalloc' > events/syscalls/sys_exit_write/trigger
```

stacktrace : Cette commande permet d'enregistrer une trace de la pile (*stacktrace*) dans le tampon de trace (*ring-buffer*) quand l'évènement déclencheur se produit. Par exemple, le déclencheur suivant enregistre la trace de la pile chaque fois que le point de trace kvm_exit est atteint avec la condition suivante exit_reason == 49 (*EPT violation*).

```
$ echo 'stacktrace if exit_reason == 49' > events/kvm/kvm_exit/trigger
```

snapshot : Cette commande permet de capturer un cliché instantané (*snapshot*) du tampon de trace et de l'enregistrer dans un fichier quand l'évènement déclencheur se produit.

```
$ echo 'snapshot if exit_reason == 49' > events/kvm/kvm_exit/trigger
```

traceon/traceoff : Ces deux commandes activent et désactivent le traçage respectivement, lorsque les évènements déclencheurs sont atteints. Avec ces commandes, nous pouvons restreindre le traçage aux périodes vraiment intéressantes à étudier.

Le déclencheur suivant permet d'arrêter le traçage après dix *EPT violations* consécutives.

```
$ echo 'traceoff:10 if exit_reason == 49' > events/kvm/kvm_exit/trigger
```

hist : Cette commande permet d'agréger les évènements dans un dictionnaire (*hashtable*) avec une valeur de clé qui prend un ou plusieurs champs de format d'évènement de trace, avec un ensemble de totaux cumulés dérivés soit par plusieurs champs d'évènements ou soit par le nombre d'évènements (*hitcount*). Le déclencheur suivant nous permet d'obtenir un histogramme des VMEXITs liés au démarrage d'une machine virtuelle.

```
$ echo 'hist:key=exit_reason:val=hitcount' > events/kvm/kvm_exit/trigger
$ cat events/kvm/kvm_exit/hist

# Histogramme: démarrage d'une machine virtuelle
{ exit_reason: 29 [ MOV DR             ] } hitcount:          1
{ exit_reason: 55 [ XSETBV             ] } hitcount:          2
{ exit_reason: 54 [ WBINVD             ] } hitcount:          6
{ exit_reason: 44 [ APIC Access        ] } hitcount:         11
{ exit_reason: 40 [ PAUSE              ] } hitcount:         83
{ exit_reason: 31 [ RDMSR              ] } hitcount:        235
{ exit_reason: 12 [ HLT                ] } hitcount:        462
{ exit_reason:  7 [ Interrupt window   ] } hitcount:        674
{ exit_reason: 52 [ Timer expired      ] } hitcount:       1431
{ exit_reason: 32 [ WRMSR              ] } hitcount:       2276
{ exit_reason:  1 [ External interrupt ] } hitcount:       3947
{ exit_reason:  0 [ NMI interrupt      ] } hitcount:      13555
{ exit_reason: 49 [ EPT misconfig      ] } hitcount:      14004
{ exit_reason: 48 [ EPT violation      ] } hitcount:      18113
{ exit_reason: 30 [ I/O instruction    ] } hitcount:     273054
{ exit_reason: 10 [ CPUID              ] } hitcount:     677671
{ exit_reason: 28 [ CR Accesses        ] } hitcount:    1381923
```

enable_hist/disable_hist : Ces deux commandes peuvent être utilisées pour contrôler le démarrage et l'arrêt des évènements attaché au déclencheur *hist*. Ces déclencheurs permettent de contrôler l'activation des agrégations générées par le déclencheur *hist*, afin d'affiner les résultats durant les investigations de cas particulier.

Ftrace est particulièrement intéressant, car il est un traceur interne au noyau Linux, contrairement aux autres traceurs Linux comme SystemTap ou Lttng. Cet avantage offre à Ftrace la possibilité de collecter les informations sur le système même durant les phases les plus sensibles. Parmi ces phases on distingue l'initialisation et l'arrêt du système.

La plupart de nos contributions qui seront présentées dans le prochain chapitre 3 ont été implémentées dans le traceur Ftrace.

2.1.2 eBPF

eBPF (*Enhanced Berkeley Packet Filter*), connu aussi sous le nom *BPF*, est originalement développé comme une technologie d'optimisation efficace pour le filtrage des paquets réseau à l'université de Berkeley en Californie [40]. Présentement, eBPF peut faire plus de choses

que juste le filtrage de réseau. Dans cette section, nous allons couvrir les fonctionnalités de traçage fournies par cet outil.

eBPF est une machine virtuelle très efficace incorporée à l'intérieur du noyau Linux. C'est un engin idéal pour le traitement rapide des évènements surgissant à haute fréquence dans le noyau. L'usage de cette technique a pris plus d'ampleur après son implantation dans le noyau Linux version 3.18. Les développeurs du noyau Linux ont rapidement utilisé eBPF dans divers systèmes du noyau, soit pour accomplir la manipulation du trafic réseau, renforcer de sécurité du noyau ou pour la surveillance du système.

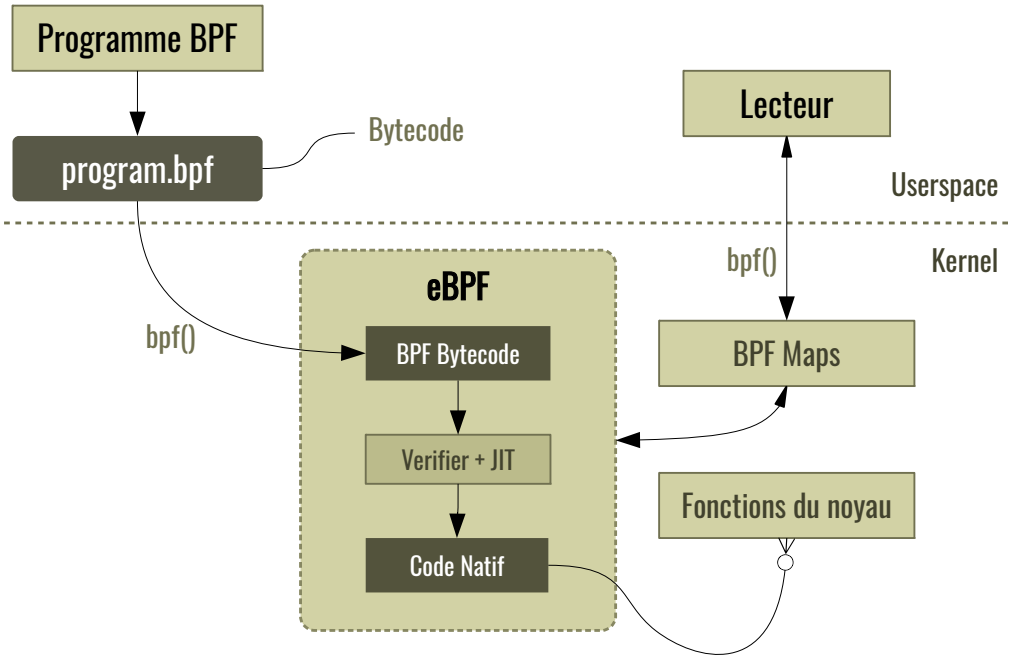

Figure 2.2 Anatomie d'un programme eBPF. Source [57]

Les programmes eBPF peuvent être définis comme des modules externes légers. Écrits en langage C, ces modules seront compilés par le Framework LLVM/Clang, transformés en code octet eBPF, puis validés par la machine virtuelle avant d'être exécutés en code natif. Le chargement d'un programme eBPF dans le noyau est fait en utilisant l'appel système *bpf syscall*. Durant cette phase, la création des dictionnaires (*eBPF-maps*) est établie. La figure 2.2 illustre toutes ses étapes.

L'utilisation de la surveillance du système (traçage) avec eBPF était possible lors de l'introduction du support de kprobes en version 4.1 du noyau Linux. Plusieurs développements ont suivi afin de supporter plus de mécanismes de traçage, comme l'attachement sur les évènements de trace et le profilage avec les évènements de Perf.

Une fois que le programme eBPF est chargé, les kprobes ou les points de trace deviennent actifs et commencent à capturer les évènements. Ces évènements peuvent soit être agrégés en utilisant les structures BPF-map, ou être redirigés vers trace pipe de Ftrace ou vers le tampon de Perf. À partir de là, les programmes d'espace utilisateur peuvent aussi administrer le programme eBPF en manipulant les structures BPF-map. La figure 2.3 illustre plus en détail les interactions.

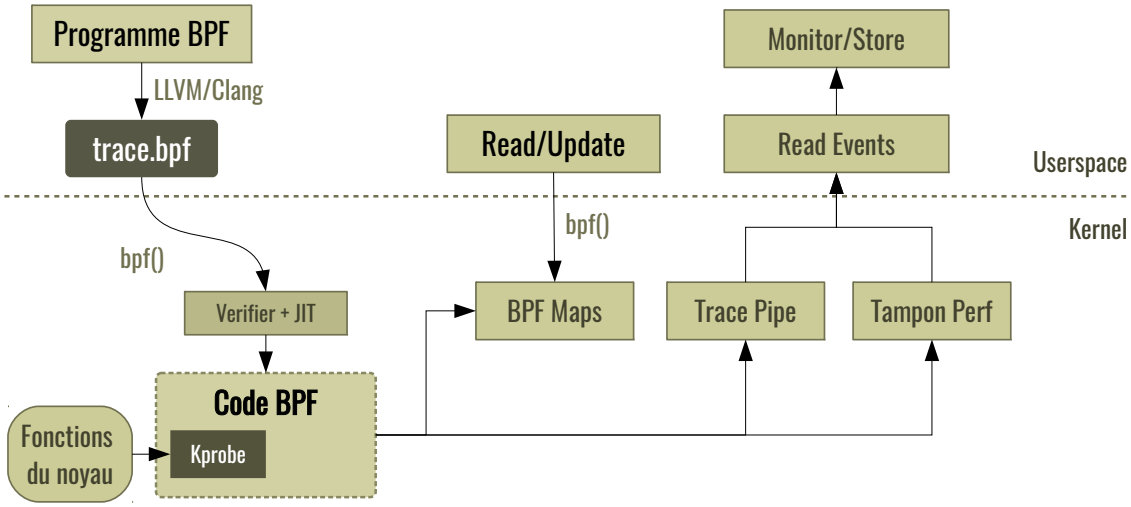

Figure 2.3 Traçage avec eBPF. Source [57]

Un exemple de programme eBPF présenté dans le listing 2.1.1, démontre comment on peut calculer de manière dynamique et en temps réel le coût associé à la virtualisation. Cet exemple utilise python comme langage d'interface *front-end* pour l'interaction avec la machine virtuelle eBPF. D'autres langages sont aussi supportés comme C++, Lua et Go, et sont maintenus par le projet IOVisor BCC (BPF Compiler Collection).

Le projet BCC est un cadriciel permettant de créer des programmes (eBPF) efficaces pour le traçage et la manipulation du noyau. Il rend le développement des programmes BPF plus facile en fournissant plusieurs outils et fonctionnalités.

Listing 2.1.1: Attacher un programme eBPF sur un point de trace

```
from bcc import BPF
b = BPF(text="""
struct kvm_data {
     u8 exit_reason;
     u64 exit_time;
 };
BPF_HASH(exit_overhead, int, struct kvm_data);
TRACEPOINT_PROBE(kvm, kvm_exit) {
    struct kvm_data data;
    int cpu_id = bpf_get_smp_processor_id();
    data.exit_reason = args->exit_reason;
    data.exit_time = bpf_ktime_get_ns();
    exit_overhead.update(&cpu_id, &data);
    return 0;
}
TRACEPOINT_PROBE(kvm, kvm_entry) {
    int cpu_id = bpf_get_smp_processor_id();
    u64 now = bpf_ktime_get_ns();
    struct kvm_data *data = exit_overhead.lookup(&cpu_id);
    if (data != NULL) {
        bpf_trace_printk("KVM_EXIT : reason=%d, cpu_id=%d, overhead=%u ns\\n",
            data->exit_reason, cpu_id, now-data->exit_time);
    }
    return 0;
};
""")
```

```
$ ./kvm_compute_overhead.py
CPU 0/KVM  KVM_EXIT reason=12 [ HLT           ], cpu_id=1, overhead=151912972 ns
CPU 0/KVM  KVM_EXIT reason=48 [ EPT violation ], cpu_id=1, overhead=914 ns
CPU 0/KVM  KVM_EXIT reason=28 [ CR accesses   ], cpu_id=1, overhead=688 ns
CPU 0/KVM  KVM_EXIT reason=52 [ Timer expired ], cpu_id=3, overhead=1703 ns
...
```

Le projet BCC offre plusieurs utilitaires d'analyse de performance prédéfinis. À titre d'exemple, nous présenterons l'outil *biolatency*. Cet outil résume la latence des E/S du périphérique de bloc sous forme d'histogramme avec un intervalle de rafraîchissement. La commande suivante présente les résultats obtenus en exécutant *biolatency* durant l'installation d'une mise à jour système [10].

```
$ biolatency
Tracing block device I/O... Hit Ctrl-C to end.
^C
        usecs               : count            distribution
        16 -> 31            : 0        |                                        |
        32 -> 63            : 23       |*****************************           |
        64 -> 127           : 11       |**************                          |
       128 -> 255           : 27       |**********************************      |
       256 -> 511           : 31       |****************************************|
       512 -> 1023          : 5        |******                                  |
      1024 -> 2047          : 17       |*********************                   |
      2048 -> 4095          : 11       |**************                          |
      4096 -> 8191          : 1        |*                                       |
```

2.2 Visualisation et analyse des traces

2.2.1 Babeltrace

Babeltrace est un logiciel libre dont le but est de traiter ou de convertir des traces de format CTF (*Common Trace Format*) [19] vers un format texte facile à lire, permettant à l'usager d'analyser visuellement la trace binaire afin d'extraire ou valider son contenu. Nous avons généré a priori une trace simplifiée présentée comme suite :

```
$ babeltrace ~/lttng-traces/simple-trace
...
[+0.87] hrtimer_expire_exit : { cpu_id = 1 }, { hrtimer=0xF4423080 }
[+0.87] hrtimer_start       : { cpu_id = 1 }, { hrtimer=0xF4423080, ... }
[+1.50] irq_softirq_raise   : { cpu_id = 0 }, { vec=7 }
[+1.16] hrtimer_expire_exit : { cpu_id = 0 }, { hrtimer=0xF4410080 }
[+0.87] hrtimer_start       : { cpu_id = 0 }, { hrtimer=0xF4410080, ... }
[+1.52] irq_softirq_entry   : { cpu_id = 1 }, { vec=1 }
[+3.03] sched_waking        : { cpu_id = 1 }, { comm="bash", target_cpu=1, ... }
[+1.55] irq_softirq_exit    : { cpu_id = 1 }, { vec=1 }
[+1.70] sched_switch        : { cpu_id = 1 }, { next_comm="bash", prev_comm="grep" }
...
```

Babeltrace Python Bindings est un utilitaire en langage *python* implémenté par dessus *babeltrace*. Il permet le développement efficace de script python pour automatiser l'analyse des traces, agréger et extraire des métriques ou présenter des résultats statistiques.

La commande suivante lit la trace et extrait les métriques relatives au démarrage du noyau :

temps total de démarrage, temps lié à chaque phase du démarrage, nombre d'évènements collectés pour chaque phase de démarrage.

```
$ ./boot-time.py ~/lttng-traces/bootup-trace
Boot time = 2608 ms
    Boot level      | time (ms) |  events  |
-------------------------------------------
early               |   93.70   |   20091  |
pure                |    2.62   |    4868  |
core                |    1.97   |    3449  |
postcore            |    1.49   |    2723  |
arch                |    0.67   |    1171  |
subsys              |  141.48   |  112486  |
fs                  |  279.30   |  497719  |
rootfs              |    6.04   |   10121  |
device              |  407.28   |  408668  |
late                |   87.86   |  133163  |
```

2.2.2 TraceCompass

Trace Compass [14] est un logiciel libre développé et maintenu en collaboration entre Ericsson et le laboratoire DORSAL de l'École Polytechnique de Montréal sous la licence d'Eclipse.

Trace Compass est un outil de visualisation et d'analyse (*offline analysis*) de traces générées par les outils de traçage. Il supporte plusieurs types de formats de trace comme CTF, Ftrace, JSON, XML et les *logs*. Il fournit des vues, des graphiques et des métriques d'une manière plus conviviale et informative pour aider les développeurs à extraire des informations utiles durant le débogage des problèmes de performance. Sa nature modulaire permet d'ajouter et d'étendre rapidement les vues graphiques existantes, en profitant du découplage des composantes, ce qui favorise plus la réutilisabilité.

À titre d'exemple, la vue de graphe d'appel présentée à la figure 2.4 montre à l'utilisateur la pile d'appels lors du démarrage du noyau Linux. On peut remarquer ceci par l'occurrence des appels aux fonctions "do_one_initcall", car les fonctions "initcall" représentent l'initialisation des modules internes chargés lors du démarrage.

L'une des fonctionnalités les plus utiles de Trace Compass est la fusion des traces et la synchronisation temporelle des vues graphiques. La figure 2.5 présente un cas réel d'utilisation de la synchronisation pour examiner la séquence d'arrêt (ou *shutdown*) du noyau d'une machine virtuelle. Sur la figure, on distingue les trois vues suivantes : vue flot de contrôle, vue des ressources et la dernière vue montre le graphe d'appel du noyau.

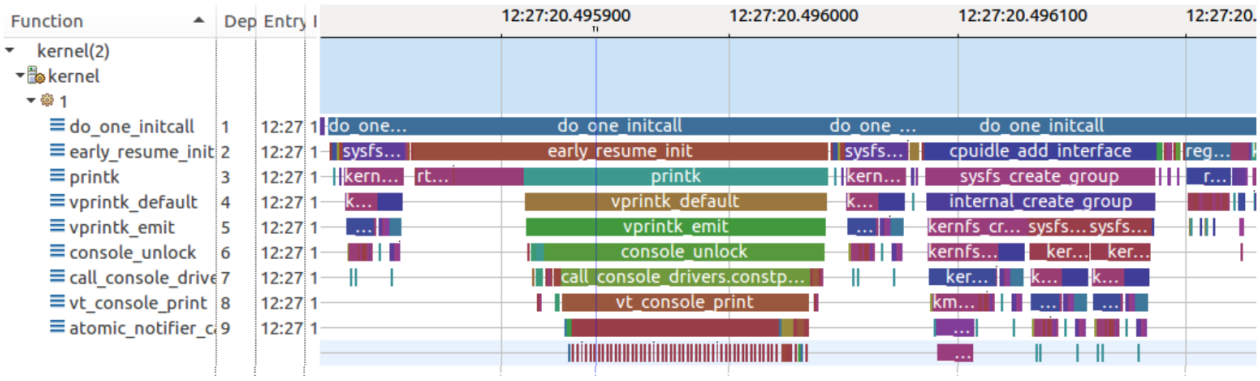

Figure 2.4 Vue de graphe d'appel durant le démarrage du noyau

La fusion des traces peut être remarquée dans la vue des ressources. Dans cet exemple, nous avons fusionné la trace du système hôte avec la trace du système invité lors de sa fermeture. La trace du système hôte ne montre aucune information relative à l'exécution du système invité, ceci revient à l'isolation imposée par la machine virtuelle. Du point de vue de l'hôte, le système invité s'exécute comme un simple processus "CPU 0/KVM". Avec la fusion de la trace du système invité, on peut ainsi voir tous les processus en exécution à l'intérieur de la machine virtuelle, et le processus "accounts-daemon" s'exécutait dans la période montrée à la figure 2.5. Plus précisément, le processus "accounts-daemon" se préparait pour l'arrêt en exécutant l'appel système "sys_exit_group".

Avec la vue de graphe d'appel (*Call Stack*), nous pouvons approfondir encore plus notre investigation. Nous pouvons analyser la pile d'appels lors de l'exécution de l'appel système "sys_exit_group". D'après la figure, nous pouvons constater que l'arrêt du processus "accounts-daemon" n'est effectué que lors de l'appel "do_exit", précédé par un appel au "zap_other_threads" qui a duré pendant 18 us. D'après la documentation du noyau Linux, l'appel "zap_other_threads" permet d'envoyer des signaux SIGKILL à tous les fils d'exécution appartenant au même groupe (*thread group*), la pile d'appels dans la figure nous montre la présence de deux appels "signal_wake_up", ces deux appels réveillent les fils d'exécution concernés pour gérer leur signal de terminaison.

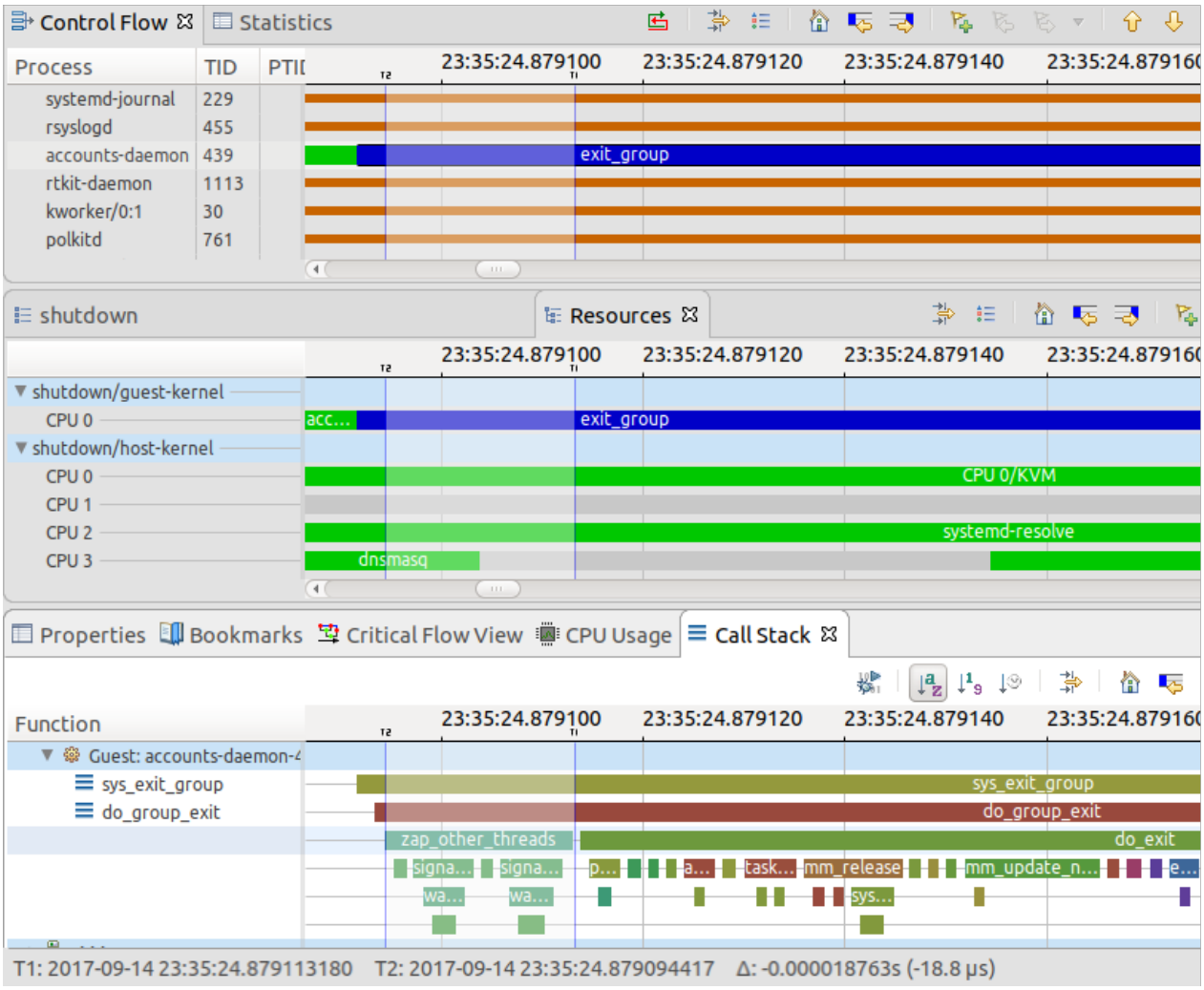

Figure 2.5 Fusion des traces, Synchronisation des vues graphiques

2.3 Virtualisation

L'origine de la virtualisation remonte à la fin des années 1960, lorsque IBM a investi beaucoup dans le développement de son nouveau produit *mainframe*. IBM a ainsi conçu le concept de virtualisation afin de partager les ressources matérielles du *mainframe* entre plusieurs usagers, avec pour but d'augmenter l'efficacité d'utilisation des ressources informatiques coûteuses.

De nos jours, plusieurs industries informatiques ont adopté la technologie de virtualisation dans leur centre de données privé. Un avantage important de la virtualisation est le fait de permettre à un environnement informatique (système hôte) d'exécuter plusieurs systèmes invités indépendants en même temps.

Les systèmes virtualisés, connus sous le nom de machines virtuelles (VMs), bénéficient de l'isolement absolu offert par la virtualisation, ceci leur donne l'illusion d'avoir le contrôle absolu et exclusif sur les ressources matérielles. Les VMs résidant sur la même machine hôte n'ont aucun accès direct au système hôte dans lequel elles s'exécutent. De plus, les parcs informatiques peuvent décider de migrer les VMs d'une machine physique vers une autre. Cette redistribution permet de réduire les coûts d'énergie en minimisant le nombre de machines physiques actives en tout temps.

D'après la définition de Popek et Goldberg [47], la virtualisation repose sur trois propriétés fondamentales :

1. **Fidélité** Un programme s'exécutant dans un environnement virtualisé doit avoir un comportement identique à celui manifesté lors de son exécution sur un système physique.
2. **Sécurité** L'hyperviseur doit être en contrôle absolu des ressources matérielles à sa disposition.
3. **Efficacité** Toutes les instructions sont exécutées directement par le matériel, sans intervention de la part de l'hyperviseur.

2.3.1 Hyperviseurs

L'hyperviseur, également appelé *Virtual Machine Monitor* (VMM), est le composant essentiel de la couche de virtualisation. L'hyperviseur est une plate-forme de virtualisation qui permet d'activer et exécuter plusieurs systèmes d'exploitation invités (ou *machine virtuelle*) simultanément sur une seule machine hôte.
Chacune de ces machines virtuelles sera capable d'exécuter ses propres programmes, car elle a l'illusion de posséder des ressources physiques comme la mémoire, le processeur et les pé-

riphériques d'E/S de l'hôte. En réalité, c'est l'hyperviseur qui alloue, multiplexe, partage et partitionne dynamiquement les ressources physiques entre toutes les VMs.

Deux types d'hyperviseurs nommés type 1 et type 2 sont largement utilisés dans la plupart des parcs informatiques. La figure 2.6 illustre l'architecture de ces hyperviseurs.

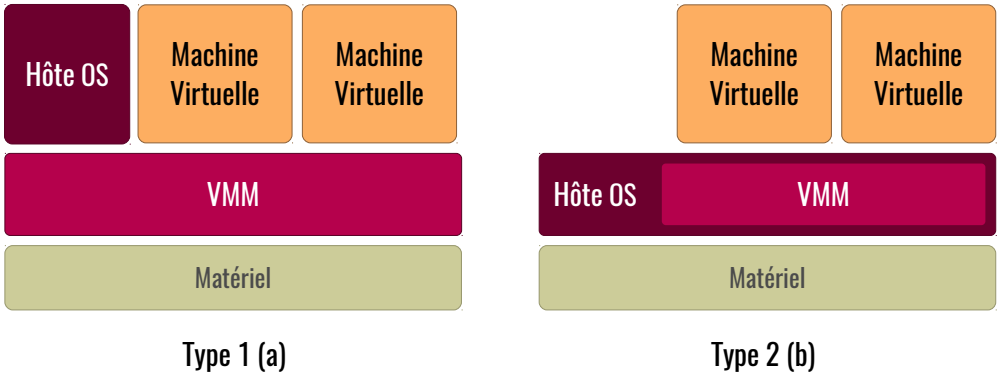

Figure 2.6 Types d'hyperviseurs

Type 1 Les hyperviseurs natifs (ou *bare metal*) sont des systèmes logiciels qui s'exécutent directement sur le matériel. Par conséquent, pour émuler et gérer une grande majorité des opérations d'entrée et de sortie, un système d'exploitation nommé domaine initial (ou Dom0) est requis. Ce dernier s'exécute sur un niveau distinct au-dessus de l'hyperviseur.
Des exemples de cette technologie classique de virtualisation sont Xen, Microsoft Hyper-V et VMWare ESX.

Type 2 Les hyperviseurs embarqués comme KVM et VMWare Server, sont conçus pour s'exécuter à l'intérieur d'un système d'exploitation traditionnel. En d'autres termes, un hyperviseur embarqué ajoute une couche logicielle distincte au-dessus du système d'exploitation hôte, et le système d'exploitation invité devient un troisième niveau de logiciel au-dessus du matériel.

Il existe différentes techniques de virtualisation pour virtualiser l'architecture x86, basée sur le niveau d'isolement fourni. Toutes exigent la présence d'un hyperviseur. La figure 2.7 compare les trois techniques côte à côte. Les sections suivantes expliqueront cette différence plus en détail.

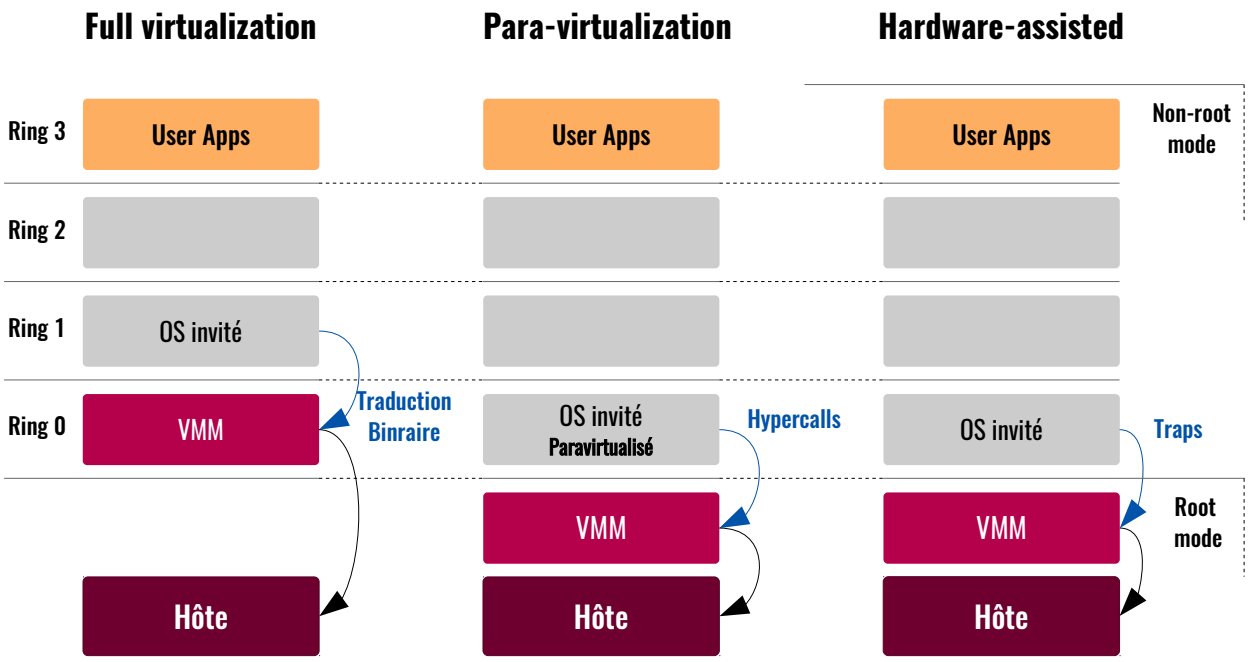

Figure 2.7 Comparaison des technologies de virtualisation. Source [38]

2.3.2 Virtualisation complète avec traduction binaire

La virtualisation complète fournit une isolation et une sécurité parfaites pour les machines virtuelles. Cette technologie simplifie beaucoup la portabilité et la migration des VMs. Ces dernières peuvent s'exécuter dans un environnement virtualisé ou natif sans modification du noyau [38].

Lors d'une virtualisation complète, aucune assistance matérielle pour virtualiser les instructions sensibles n'est nécessaire. L'hyperviseur effectue la traduction binaire de toutes les instructions du système d'exploitation à la volée (*on the fly*). Le traducteur binaire (BT) n'optimise pas le flux d'instructions binaires. Le flux d'instructions qui a été traduit est conservé dans une cache dans le Translator Cache (TC), pour une utilisation future. Dans le cas d'une boucle, la traduction ne sera effectuée qu'une seule fois.

Les applications utilisateur ne seront pas touchées par le traducteur binaire (BT). Le code utilisateur est considéré comme "sûr" (*safe*). Cet aspect permet ainsi aux applications en mode utilisateur de s'exécuter directement sur le processeur comme si elles fonctionnaient nativement [16].

Le traducteur binaire (BT) de VMware est le premier à avoir virtualisé l'architecture x86 à faible surcoût. Le BT est léger comparé à celui utilisé par Intel Itanium. VMWare n'a pas eu

besoin de traduire d'une architecture ISA (Instruction Set Architecture) à une autre, mais s'est basé seulement sur un traducteur x86 à x86.

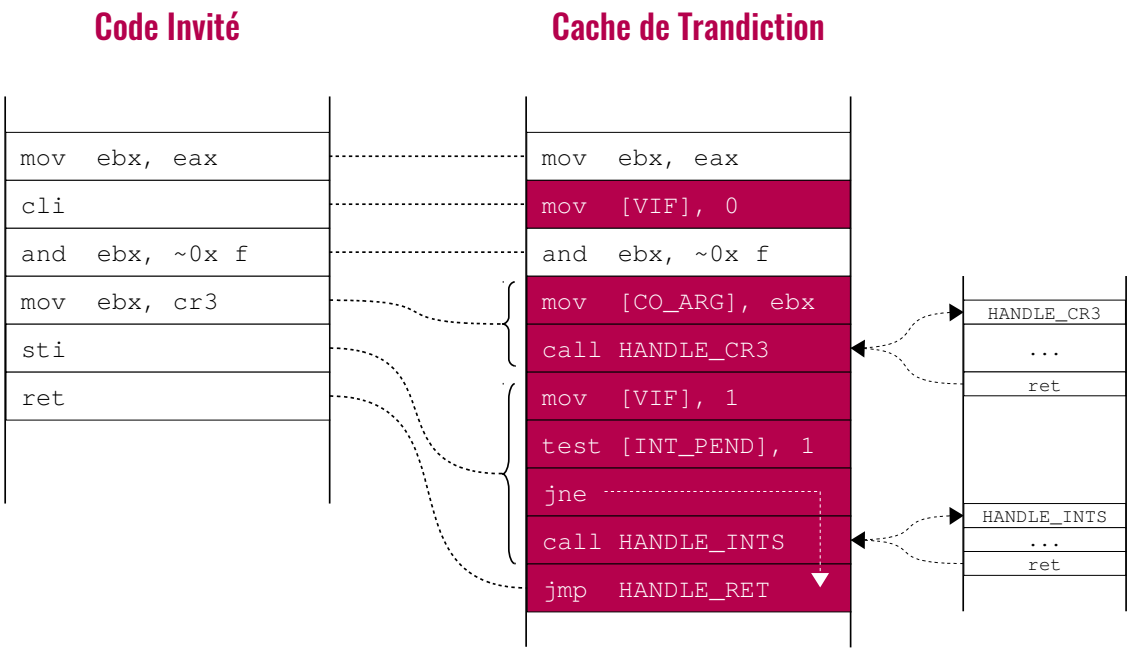

Figure 2.8 Exemple de traduction binaire de x86 vers x86 virtualisé. Source [60]

Lors de la présence d'instructions privilégiées dans le code, ou de l'accès au matériel physique, le BT doit rendre le code noyau traduit plus long que l'original. Le BT remplacera ce code binaire par du code sécuritaire (*safe*) qui manipule une zone mémoire. Par sécuritaire, nous voulons dire sans danger pour les autres VMs et l'hyperviseur. La figure 2.8 illustre un exemple de traduction d'instructions privilégiées, telles que la modification des registres sensibles comme CLI ou CR3. Ces accès aux registres seront traduits par de simples écritures en mémoire. Dans la plupart des cas, la traduction est souvent une copie exacte de l'original, car les instructions privilégiées ont une faible présence par rapport aux instructions normales.

2.3.3 Virtualisation assistée par matériel

Les fournisseurs de matériel Intel et AMD ont adopté rapidement la virtualisation, et introduit de nouvelles extensions Intel-VT (Virtualization Technology) et AMD-V aux processeurs Intel et AMD respectivement, afin de simplifier les techniques de virtualisation.

Les extensions VT et SVM ont rendu la virtualisation classique (proposée par Popek et Goldberg) possible sur l'architecture x86. L'utilisation de la méthode *trap and emulate* permet une virtualisation native et complète de l'architecture x86. Ces extensions matérielles consti-

tuent un pas d'évolution important vers la suppression de la nécessité d'utiliser la traduction binaire ou la paravirtualisation [38].

La performance résultante de cette technologie dépend principalement de la fréquence des sorties (*trap*). Chaque sortie correspond à une exception générée par le système invité. Lors de chaque sortie, l'état de sortie du système invité est enregistré en mémoire dans une structure de donnée nommée VMCS (Virtual Machine Control Structure). Les sorties sont des changements de contexte de VM vers l'hyperviseur, souvent très coûteux. La réduction de la fréquence des sorties est l'optimisation la plus importante pour améliorer la performance de la virtualisation [2].

Le système invité s'exécute dans un mode nommé le mode invité (*user mode*). Le code invité, qu'il s'agisse du code de l'application ou du code noyau, s'exécute en mode invité. Lors de certains évènements, comme les instructions privilégiées, le processeur quitte le mode invité et passe en mode hôte (*root mode*), cette transition est connue comme VMEXIT. L'hyperviseur s'exécute en mode hôte, ce dernier détermine la raison de la sortie en lisant le contenu de VMCS. Ensuite, il exécute les actions requises pour effectuer l'émulation, puis retourne le système invité en mode invité par l'instruction **VMRUN** (VMLAUNCH ou VMRESUME) [28].

2.3.4 Paravirtualisation

La paravirtualisation est un mécanisme de communication entre l'hôte et ses invités pour but l'amélioration de la performance. L'efficacité de cette communication est réalisée par la coopération du système d'exploitation invité avec l'hôte, au moyen d'une interface permettant au système invité d'indiquer son intention à l'hyperviseur.

De nombreuses techniques de paravirtualisation sont utilisées actuellement dans Linux, on peut les grouper sous deux catégories différentes : paravirtualisation des périphériques E/S et l'hypercall.

Hypercall

Lorsque le système d'exploitation invité s'exécute en mode utilisateur, toutes les instructions sensibles exécutées doivent générer un changement de contexte (*Trap*) vers l'hyperviseur. Pour résoudre ce problème, en utilisant l'approche de paravirtualisation, le code source du système d'exploitation invité sera modifié afin de remplacer ces instructions sensibles par une seule invocation à l'hyperviseur par l'intermédiaire de l'hypercall, afin que le travail lourd soit effectué du côté hôte.

L'hypercall est principalement une demande spéciale du système invité au système hôte pour effectuer une opération privilégiée, et il peut également être utilisé comme mécanisme de communication entre les deux parties, tel qu'illustré à la figure 4.3.

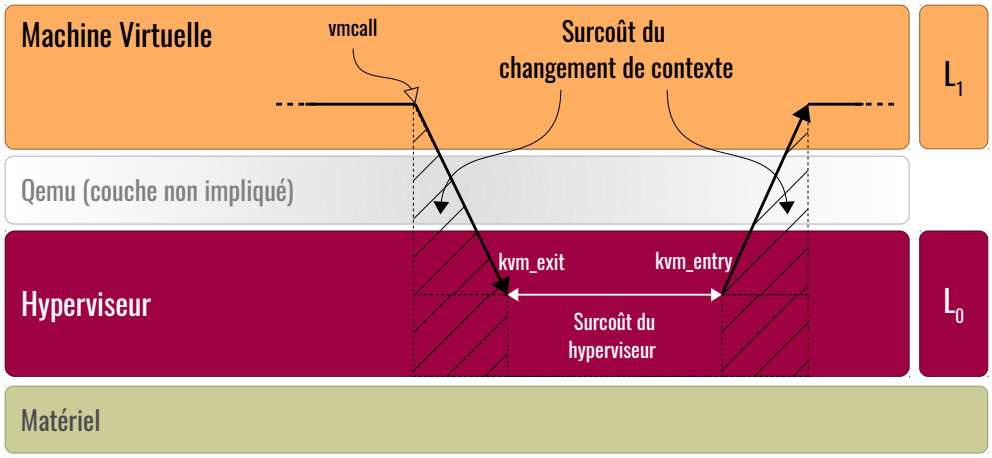

Figure 2.9 Anatomie d'un hypercall

Listing 2.1 Exemple d'appel hypercall en C.

```
#define do_hypercall(nr, p1, p2, p3, p4) \
__asm__ __volatile__(".byte 0x0F,0x01,0xC1\n":: \
        "a"(nr), "b"(p1), "c"(p2), "d"(p3), "S"(p4))

int main(int argc, char** argv)
{
    do_hypercall(99, 100, 101, 102, 103);
    return 0;
}
```

Le mécanisme de transport de l'hypercall est effectué à l'aide de registres. À titre d'illustration, pour l'architecture x86, le système invité peut utiliser les registres par défaut généraux **ax**, **bx**, **cx**, **dx** et **si** pour placer les données avant d'émettre l'instruction *vmcall* comme illustré dans le listing 2.2. Le registre **ax** sera réservé pour le numéro de l'hypercall pendant la soumission, puis l'hyperviseur utilisera **ax** pour placer la valeur retournée lors du retour au mode utilisateur [48]. Avec l'architecture 64 bits, des registres supplémentaires peuvent être utilisés, tels que **r8** jusqu'à **r15**, couplés avec un mécanisme de *clobbering* du compilateur approprié.

Listing 2.2 Code assembleur d'un hypercall.

```
88f <main>:
88f:   55                push   %rbp
890:   48 89 e5          mov    %rsp,%rbp
893:   53                push   %rbx
894:   89 7d f4          mov    %edi,-0xc(%rbp)
897:   48 89 75 e8       mov    %rsi,-0x18(%rbp)
89b:   b8 63 00 00 00    mov    0x63,%eax
8a0:   bf 64 00 00 00    mov    0x64,%edi
8a5:   b9 65 00 00 00    mov    0x65,%ecx
8aa:   ba 66 00 00 00    mov    0x66,%edx
8af:   be 67 00 00 00    mov    0x67,%esi
8b4:   89 fb             mov    edi,%ebx
8b6:   0f 01 c1          vmcall
8b9:   5b                pop    %rbx
8ba:   5d                pop    %rbp
8bb:   c3                retq
```

Un cas pratique d'utilisation de l'hypercall est disponible dans le projet Linux. L'hypercall peut être utilisé par le système invité pour réveiller un CPU virtuel (vCPU) à partir de l'état d'arrêt (HLT). Dans le cas où une tâche donnée est en attente d'accéder à une ressource, la tâche effectuerait une interrogation active (*spinlock*) sur un vCPU. La paravirtualisation peut être utilisée dans ce cas pour économiser les cycles de CPU gaspillés, en exécutant l'instruction HTL lorsqu'un seuil d'intervalle de temps s'est écoulé.

L'exécution de l'instruction HLT provoque un changement de contexte vers l'hyperviseur, puis le vCPU sera mis en mode veille jusqu'à ce qu'un évènement approprié soit déclenché. Lorsque la ressource est libérée par une tâche dans un autre CPU virtuel, la tâche peut réveiller le vCPU en cours de sommeil en déclenchant l'hypercall connu comme KVM_HC_KICK_CPU [48].

Paravirtualisation des E/S

Dans la paravirtualisation d'E/S, le système invité est conscient de la présence de la couche de virtualisation. Il charge donc les pilotes (*front-end*) appropriés pour coopérer avec l'hyperviseur, qui à son tour implémente les pilotes back-end pour l'émulation de ces périphériques. La paravirtualisation d'E/S est basée sur l'utilisation de mémoire partagée. Deux techniques de partage de mémoire sont connues : partage de pages et partage de régions mémoires.

a) *Page sharing*

Virtio, initialement développé par Rusty Russell [56], est un cadriciel de paravirtualisation qui fournit une couche d'abstraction de transport (*virtqueue*) par-dessus les périphériques matériels. Il offre une interface de développement commune pour réduire la duplication de code et uniformiser les pilotes de périphériques virtuels entre les différents hyperviseurs. De plus, la même interface des pilotes *front-end* peut communiquer avec les pilotes *back-end* de différents hyperviseurs. La figure 2.10 illustre cette architecture.

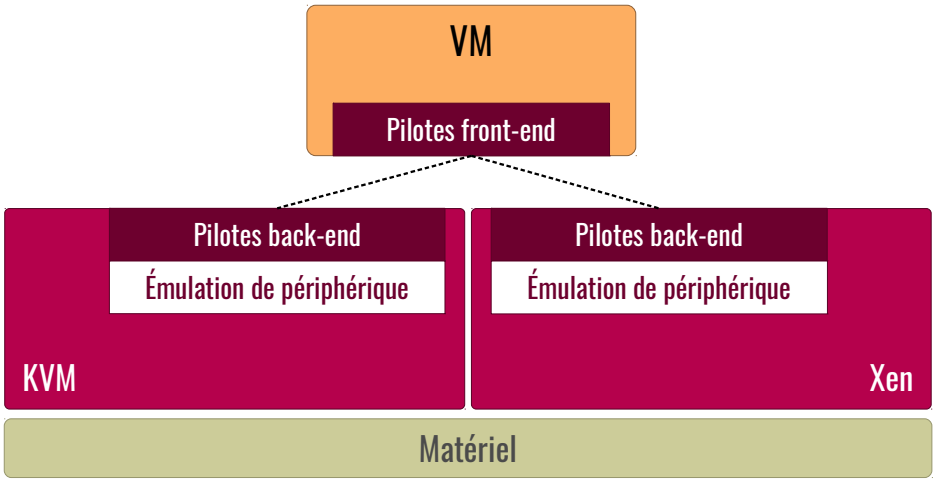

Figure 2.10 Abstraction des pilotes avec Virtio. Source [33]

Virtio est basé sur des descripteurs de tampon, qui sont des listes de pointeurs d'adresse physique du système invité. Les opérations comme l'ajout de ces descripteurs dans une file d'attente (*virtqueue*) ou la signalisation à l'hyperviseur pour lire le tampon sont utilisées pour communiquer avec le système hôte.

Le noyau Linux inclut actuellement plusieurs pilotes Virtio, tels que le pilote du réseau nommé virtio-net, le pilote de périphérique de bloc virtio-blk, ainsi que plusieurs autres pilotes comme virtio-console, virtio-vsock, virtio-scsi, virtio-balloon, virtio-pci, virtio-crypto et virtio-trace.

b) *Memory Sharing Region*

Nahanni est une implémentation de la technique *Memory Sharing Region*. Cet outil est développé par Cameron Macdonell [36] dans QEMU sous la forme d'un canal de communication basé sur la technologie de mémoire partagée POSIX à haute bande passante. Cet outil permet une communication hôte-invités ainsi qu'inter-VMs à faible latence. Ce mécanisme ne nécessite aucune intervention de la part de l'hyperviseur. De plus, les opérations de lecture

et écriture sont effectuées sans copie.

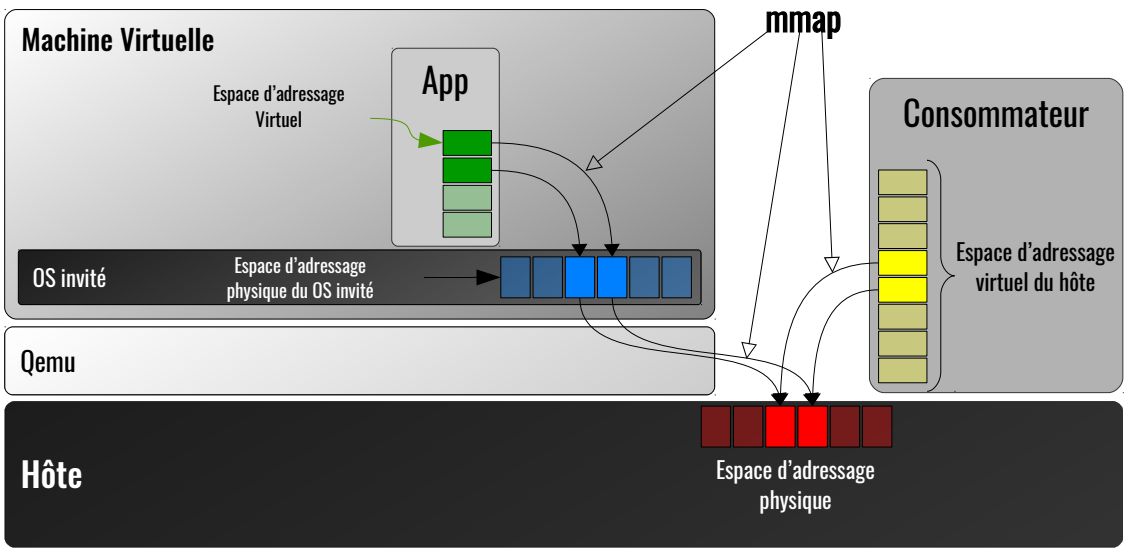

Figure 2.11 Architecture de mémoire partagée avec zéro-copie

Nahanni est implémenté en tant que nouveau périphérique PCI (*Peripheral Component Interconnect*) virtuel dans QEMU. Ce dernier sera utilisé pour partager la région de mémoire POSIX de l'hôte (qu'il a rendue disponible dans QEMU) avec le noyau invité. Comme illustré dans la figure 2.11, les applications d'espace utilisateur à l'intérieur du système d'exploitation invité peuvent accéder directement à la zone de mémoire partagée en chargeant le périphérique à l'aide de l'appel système mmap. Ce dernier affichera la zone de mémoire dans leur espace d'adressage avec le mécanisme de zéro-copie. Nahanni prend également en charge le mécanisme de notification, afin d'informer l'hôte ou d'autres systèmes invités de la disponibilité des données, Nahanni utilise des descripteurs de fichiers d'évènements (*eventfds*) comme mécanisme de signalisation et de synchronisation.

2.4 Hypertraçage

L'hypertraçage (ou *hypertracing*), est un mot composé de deux mots, le premier est "hyper" qui fait référence à l'hyperviseur, le deuxième est "traçage" qui fait référence à la journalisation. Nous avons vu la nécessité d'introduire ce nouveau mot pour faire référence à un nouveau domaine qui s'est développé durant la dernière décennie.

La couche d'isolation des machines virtuelles limite l'efficacité du traçage, les informations collectées sont insuffisantes (voire limitées) pour la compréhension des évènements surgissant sur le système. L'hypertraçage est une méthode de traçage adaptée pour l'investigation des problèmes dans un environnement virtuel. La collaboration des systèmes hôte et invité est primordiale pour activer l'hypertraçage. Cette collaboration exige une communication hôte-invité ou invité-invité pour le partage d'information qui manque, où l'hyperviseur joue un rôle important.
Selon la technique utilisée pour le partage de données, l'hypertraçage peut être effectué en deux modes : synchrone ou asynchrone. La figure 2.12 présente l'architecture d'hypertra-

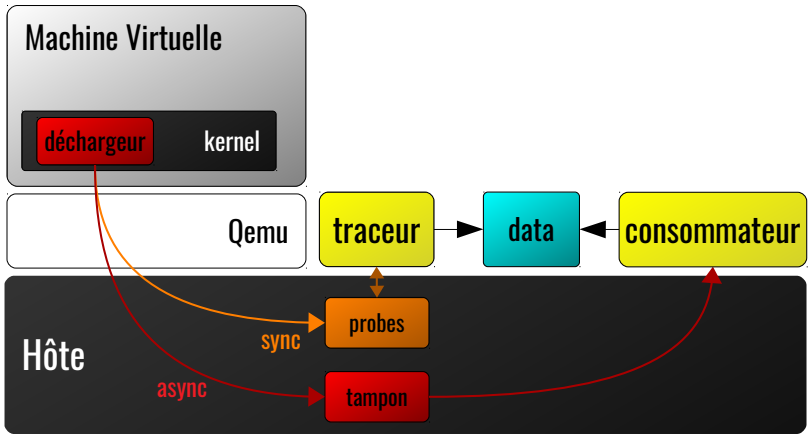

Figure 2.12 Architecture d'hypertraçage

çage pour les modes synchrone ou asynchrone. Le déchargeur installé dans le système invité peut choisir quel mode utilisé. Chaque mode exige une technique de paravirtualisation pour l'effectuer. L'hypercall se comporte comme un *syscall* et est donc de nature synchrone. Les mémoires partagées sont de nature asynchrone, car elles utilisent des tampons qui seront consommés une fois remplis.

Plusieurs outils ont été développés au cours des dernières années. Les prochaines sous-sections présentent plus en détail certains outils pouvant être classés comme outils d'hypertraçage :

2.4.1 IVRing

IVRing est un canal de communication basé sur un tampon partagé entre le système hôte et le noyau invité. Il est conçu en utilisant le périphérique virtuel de mémoire partagée *ivshmem* [36].

IVRing est un tampon qui peut être utilisé pour la journalisation ou pour le traçage des machines virtuelles. Les données enregistrées depuis la VM peuvent être partagées en temps réel avec l'hôte ou avec d'autre VMs, sans recours à la copie. Cet outil a été développé en 2012 sous forme d'un pilote de périphérique pour les systèmes invités, puis proposé à la communauté de QEMU [5] et du noyau Linux. Cependant, la communauté ne l'a pas intégré, étant donné l'existence de deux implémentations redondantes du tampon dans le noyau Linux (celui de Ftrace et Perf). L'intégration d'un troisième tampon ajouterait plus de duplication.

2.4.2 Qemu hypertrace

Qemu hypertrace est un canal de communication établi entre le système invité et QEMU. Il se constitue de trois sous-canaux distincts : données, configuration et contrôle. Il permet aussi d'émettre des évènements dans QEMU en utilisant son infrastructure de traçage. Pour récupérer ces évènements, il faut activer le traçage dans l'espace d'utilisateur.

La principale utilité d'hypertrace réside dans son utilisation pour le débogage, et l'association des évènements surgissant dans QEMU avec l'exécution du système invité, afin d'identifier les "régions d'intérêt" dans le code invité. Ces régions seront utilisées par la suite pour suivre le comportement de QEMU à l'égard de leur exécution, en optimisant grandement la performance en n'activant que les évènements de traçage à l'intérieur des régions d'intérêt.

L'hypertrace permet aussi d'ajouter des marqueurs nommés "point de progrès" dans le code invités, permettant une corrélation facile entre les divers évènements de trace dans QEMU et dans le système invité.

On peut utiliser les régions d'intérêt et les points de progression sur le code invité pour investiguer les problèmes de performance du traducteur binaire de QEMU TCG (Tiny Code Generator). Chaque évènement émis par l'invité sera émis en utilisant le *timestamp* de l'hôte, ceci évitera les problèmes de désynchronisation des horloges, et facilite la comparaison des temps d'exécution de l'hôte par rapport au code invité intéressant.

La figure 2.13 montre l'architecture de QEMU hypertrace. Ce dernier offre trois champs de mémoire (canaux) de communication, dont deux (commande, données) sont importants. Le canal de commande est utilisé par l'invité pour signaler à QEMU que de nouvelles données dans le canal de données sont prêtes pour le traitement. Les écritures sur le canal de

commande sont bloquantes, car elles sont interceptées par QEMU. Le canal de données est un tampon de mémoire standard utilisé par l'invité pour écrire les évènements qui seront consommés par QEMU, une fois le canal de commande sera utilisé.

Les deux canaux supportent l'option de décalage des régions mémoires utilisées. Cette option permet d'avoir plusieurs VMs ou plusieurs fils d'exécutions ou processeurs qui utilisent le canal hypertrace sans avoir à se synchroniser. Cette méthode est similaire aux implémentations des périphériques virtuels dans SR-IOV (Single Root I/O Virtualization). SR-IOV permet à un périphérique PCIe (PCI Express) d'apparaître comme plusieurs périphériques PCIe physiques distincts, et ainsi de se présenter sous la forme de plusieurs instances multiplexées et partagées à travers les VMs.

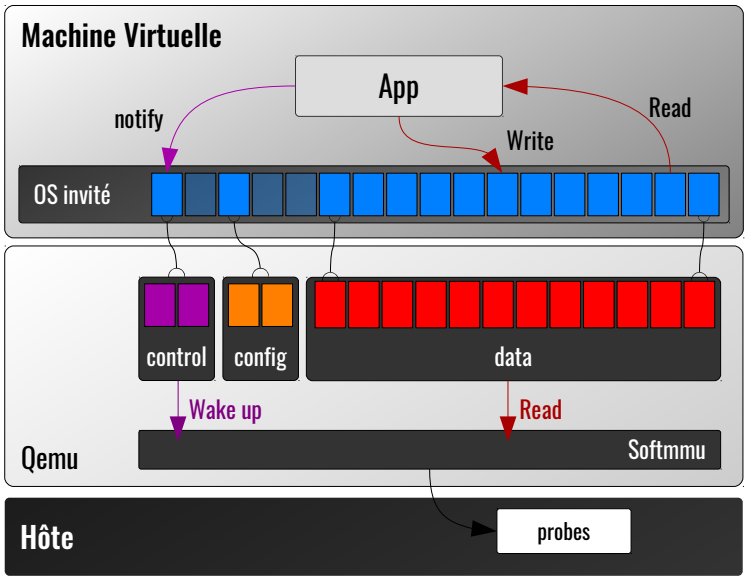

Figure 2.13 Architecture de QEMU hypertrace

2.4.3 Virtio-trace

Virtio-trace [61], introduit au noyau Linux par Yunomae, est un système de surveillance à faible surcoût permettant de collecter les évènements de trace du noyau directement depuis le système hôte à travers virtio-serial. Ce mécanisme permet d'éviter le surcoût de performance à l'utilisation de la couche réseau lors de l'envoi d'évènements. Plus précisément, un agent à l'intérieur du système invité effectuera les opérations d'épissage (*splice*) sur le tampon (*ring-buffer*) de Ftrace lorsque le traçage est activé. Ce mécanisme permet un transfert de données efficace vers le tampon virtio-ring de QEMU, qui sera ensuite consommée directement par l'hôte, sans faire de copie de mémoire.

2.4.4 Xenrelay

Jin at al. [32], ont développé un outil de paravirtualisation nommé Xenrelay. Cet outil est un mécanisme de transmission unidirectionnel unifié de transmission entre VM, très efficace pour transférer des petites données de trace fréquemment générées, du domaine invité vers le domaine privilégié. Xenrelay a la capacité de relayer des données sans verrou ni notification. Ce mécanisme a été implémenté en utilisant un tampon circulaire producteur-consommateur avec un mécanisme de mappage pour éviter d'utiliser la synchronisation. Cet outil a permis d'optimiser les performances de la virtualisation de l'hyperviseur Xen, en facilitant le suivi et l'analyse des problèmes de virtualisation.

2.4.5 XenMon

Gupta at al. [29] ont développé XenMon, une infrastructure de monitorage et de profilage des performances des machines virtuelles sur Xen. Cet outil rapporte l'utilisation des ressources des machines virtuelles et fournit un aperçu supplémentaire de l'accès aux ressources partagées et aussi de l'ordonnancement des ressources dans Xen.

XenMon signale une variété de métriques (CPU usage, blocked time, waiting time) qui sont accumulées lors d'exécution d'une machine virtuelle sur le processeur. Ceci est effectué dans un intervalle de temps qui est configurable. XenMon rapporte aussi d'autres métriques plus avancées, comme le temps d'allocation et le nombre d'E/S. Le temps d'allocation représente combien de temps CPU a été alloué au domaine par l'ordonnanceur de Xen, tandis que le nombre d'E/S indique le nombre de transferts de page (*flip*) effectués lors des opérations E/S entre les domaines invités et le domaine privilégié.

2.4.6 Xenoprof

Menon at al. [42] présente Xenoprof, un outil de profilage statistique implémenté pour l'environnement des machines virtuelles sur Xen. Cet outil permet de collecter des échantillons périodiques de données de performance depuis les évènements matériels, en effectuant un profilage organisé de plusieurs machines virtuelles co-localisées sur le même système. Ces évènements matériels sont les cycles d'horloge, mémoire cache et TLB. Ils sont aussi aussi utilisés par d'autres traceurs comme Perf [17]. Xenoprof est développé afin d'analyser les dégradations de performance induites par des applications catégorisées comme utilisateur intensif du réseau, qui s'exécutent dans les domaines Xen.

Xenoprof utilise le traceur OProfile pour gérer le profilage de performance des domaines depuis l'hyperviseur via des hypercalls. Xenoprof fournit des échantillons à la couche OProfile

en utilisant le mécanisme d'interruption virtuelle de Xen.

2.4.7 Dtrace-virt

Dtrace-virt [59] est une extension implémentée pour le traceur Dtrace dans le système d'exploitation FreeBSD. C'est un outil de surveillance spécialement conçu pour surveiller les machines virtuelles non fiables. Dtrace-virt a été conçu comme un outil de détection d'intrusion et d'analyse de logiciels malveillants (ou *malwares*). Cet outil a été conçu pour détecter l'injection de code malveillant en envoyant les évènements de trace agrégés en utilisant le mécanisme hypercall comme moyen de communication rapide avec l'hyperviseur. L'approche de l'hypercall est avantageuse, au lieu d'utiliser les protocoles réseau, traditionnellement utilisés par de nombreux outils de traçage distribués, qui induisent une surcharge très importante dans un environnement virtualisé.

Les auteurs n'ont fourni aucun cas d'utilisation pour évaluer l'utilité de leur approche pour la détection des problèmes de sécurité. Les auteurs discutent seulement des problèmes de conception et de mise en œuvre de l'infrastructure de Dtrace-virt pour surveiller les machines virtuelles.

2.4.8 LgDb

LgDb [34] est un outil de débogage de machine virtuelle qui utilise la plateforme Lguest pour fournir un environnement de développement et de tests du noyau, ainsi que le profilage et la couverture de code noyau. Le mécanisme utilisé par LgDB est similaire aux débogueurs traditionnels. L'environnement provoquera un changement de contexte du mode invité au mode hôte, afin d'arrêter l'exécution de l'invité lorsque le flux de code atteint un point spécifique. Deux approches ont été proposées à cet effet, la première étant l'approche hypercall, celle-ci nécessitant une instrumentation manuelle du code noyau inspecté pour définir les points de rupture. La deuxième approche est basée sur le débogueur, aucune modification du code n'est alors nécessaire. Cette approche utilise le débogueur du noyau (KGDB) pour activer des points d'arrêt (*breakpoints*) à travers le port série virtio.

2.4.9 Résumé des outils d'hypertraçage

Cette section présente un sommaire comparatif des outils d'hypertraçage précédemment présentés. Le tableau 2.1 présente les résultats.

Tableau 2.1 Comparaison des outils d'hypertraçage

	Hypercall		Memory Sharing Region		Page Sharing			
	Dtrace-virt	LgDb	IVRing	Qemu hypertrace	Virtio-trace	XenMon	Xenoprof	Xenrelay
Domaine d'application	Détection d'injection de code malveillant	Tests des modules noyau	Eviter d'envoyer les traces par réseau	Traçage de TCG	Traçage des périphé-riques virtuels	App. au E/S intensives	App. à réseau intensif	Traçage de la virt. d'E/S
Type de surveillance	Profilage	Débogage	Traçage	Traçage	Traçage	Profilage	Profilage	Traçage
Mécanisme de partage	vmcall	vmcall	Nahanni : PCI, mmap, UIO, eventfds	Mémoire Qemu : PCI, mmap	Virtio	Grant Table	Grant Table	Grant Table
Hôte-invité/ invité-invité	Hôte-invité	Hôte-invité	Les deux	Qemu-invité	Les deux	Inter-domaine	Inter-domaine	Inter-domaine
Implication d'hyperviseur	Fort	Moyen	-	-	Faible	Faible	Faible	Faible
Type de communication	Sync	Sync	Async	Async	Async	Async	Async	Async
Plateforme	KVM based					Xen based		

Dans ce chapitre, nous avons commencé par introduire le traçage dans Linux et les différents outils d'analyse de trace présentement utilisés. Ensuite, nous avons expliqué les différentes techniques de virtualisation, et nous avons mis plus d'emphase sur la technique de paravirtualisation, puisqu'elle sera utilisée dans nos contributions. Puis, nous avons introduit l'hypertraçage, une méthode récemment établie pour investiguer les problèmes survenus dans des systèmes virtuels. Pour finir, nous avons résumé les différents outils d'hypertraçage dans un tableau comparatif.

Dans le prochain chapitre 3, nous allons présenter les choix de conception et les différentes contributions menant à la réalisation de ce projet de recherche.

CHAPITRE 3 MÉTHODOLOGIE

Nous présentons dans cette section les aspects méthodologiques et les choix de conception qui ont conduit au développement de ce projet de recherche. Nous commençons par établir le contexte du travail ainsi que l'environnement dans lequel nous avons effectué nos expérimentations. Par la suite, nous présentons les détails sur les approches d'hypertraçage proposées pour relever les défis soulevés dans le chapitre 1 "Introduction". Pour finir, nous présentons les différentes contributions apportées aux logiciels libres KVM, Ftrace et TraceCompass, menant à la réalisation de l'article présenté dans le chapitre 4.

3.1 Contexte de travail

Dans ce projet de recherche, QEMU/KVM est utilisé comme technologie de virtualisation, dans un environnement Linux. Dans QEMU, un système invité est représenté simplement par un processus QEMU régulier. Chaque système invité est constitué de deux types de fils d'exécution, un premier fil d'exécution *vCPU* par CPU virtuel et un autre fil d'exécution *iothread* dédié au traitement des opérations d'E/S telles que les paquets réseau et opérations des disques. Plusieurs VMs résident sur la même machine hôte et s'exécutent sans connaître l'existence des autres. De plus, des applications hôte comme Google Chrome ou nginx sont également en concurrence avec QEMU pour les mêmes ressources physiques.

L'approche traditionnellement utilisée lors des investigations de problèmes de virtualisation exige l'activation du traçage à tous les niveaux de virtualisation, telle qu'illustrée à la figure 3.1. De plus, la gestion du temps est une responsabilité propre à chaque système d'exploitation, il est essentiel d'activer aussi les méthodes de synchronisation pour réduire le décalage entre ces systèmes tracés. Cependant, l'activation de la synchronisation vient avec un surcoût de performance, et la précision obtenue peut être insuffisante lors d'investigations de problèmes ardus.

Dans certains cas, comme lors de l'initialisation ou de l'arrêt du système. L'enregistrement sur disque ou l'envoi par réseau (des évènements de traces) n'est pas possible. Pour investiguer les problèmes survenant dans de telles situations, il est primordial d'utiliser les techniques de partage d'information hôte-invité (ou *hypertracing*), adaptées aux problèmes investigués.

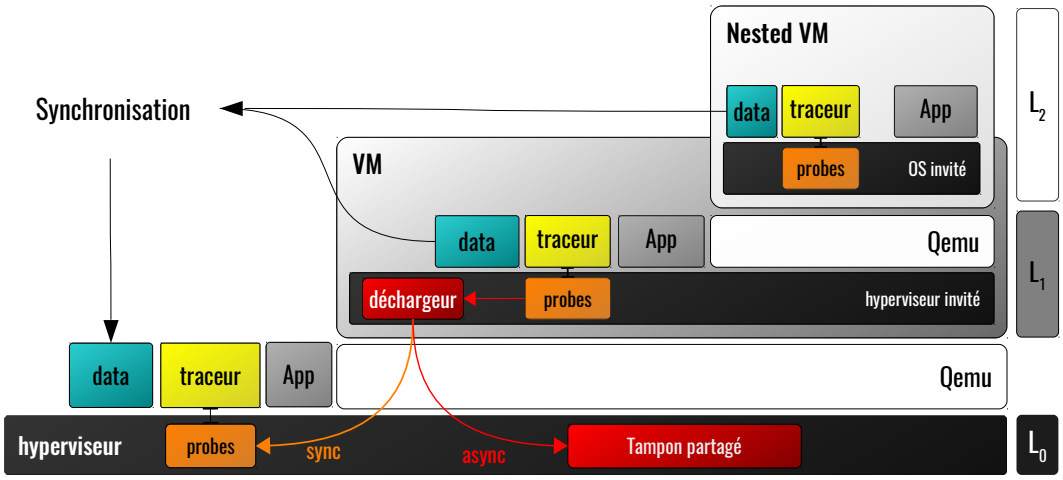

Figure 3.1 Architecture d'hypertraçage requise

3.2 Approche utilisée pour l'hypertraçage

L'article du chapitre 4 explique plus en détail le concept de l'hypertraçage, la présente section fournit seulement un bref récapitulatif pour l'introduire.

3.2.1 Hypercall

Les outils courants pour le traçage et l'analyse de performance ne sont généralement pas disponibles pendant l'initialisation du système. Pour cette raison, l'analyse du comportement pendant la phase de démarrage présente des défis supplémentaires. Dans les systèmes virtualisés, le système hôte est pleinement disponible, mais n'a pas accès à toute l'information. De son côté, le système virtuel a toute l'information, mais est en phase d'initialisation.

À fin de pouvoir investiguer les problèmes qui arrivent durant le démarrage d'un système d'exploitation invité dans le contexte infonuagique, on utilise des techniques de traçage. Toutefois, la collection des évènements durant le démarrage est très difficile. Par exemple, la collection des évènements exige d'avoir un grand tampon de mémoire vive, nécessaire pour éviter la perte des évènements.

Pour contourner toutes ces limitations, on propose une méthode de paravirtualisation qui sera implantée directement dans le noyau invité. Cette technique permet d'envoyer continuellement les évènements du démarrage depuis le noyau invité vers la machine hôte, en utilisant l'interface de virtualisation hypercall. La performance de cette technique sera évaluée et différents mécanismes pour efficacement activer les points de trace, filtrer les données et passer des arguments de différentes tailles seront proposés, examinés et évalués.

3.2.2 Mémoire partagée

Les outils courants pour le traçage et l'analyse de performance ne peuvent pas mesurer les latences causées par la couche de virtualisation sans avoir accès au système hôte. Pour remédier à cela, nous avons développé une technique d'hypertraçage à l'aide d'une mémoire partagée [36]. Le système hôte peut adapter cette technique pour offrir aux systèmes invités plus de visibilité sur les évènements qui surgissent au niveau de la couche de virtualisation, mais sont invisibles aux invités. Une fois ce mécanisme mis en place, les outils de traçage courants peuvent à présent collecter les informations partagées par l'hôte, et des analyses de performance peuvent ainsi être développées. La fusion de l'information envoyée par l'hôte et le système invité devra tenir en considération la différence de source d'horloge utilisée par ces systèmes. L'ordre chronologique des évènements est primordial pour l'investigation des problèmes de performance.

3.3 Contributions pertinentes

3.3.1 Contributions aux traceurs Ftrace

Ftrace est un traceur interne au noyau Linux, cette propriété intéressante donne la possibilité d'utiliser ce traceur pour investiguer les problèmes au démarrage du noyau.

Very Early Function Tracing : Nous avons proposé cette contribution pour activer le traçage de fonctions très tôt lors du démarrage. La figure 3.2 montre les changements faits pour permettre de tracer très tôt lors du démarrage du noyau.

La fonction start_kernel() représente le point d'entrée du noyau Linux, on peut remarquer que la plupart des fonctions appelées par start_kernel sont soit **_init* pour initialisation ou *setup_** pour configuration. L'ordre des appels d'initialisation est très important durant la phase de démarrage. Ces appels d'initialisation sont tous exécutés en série par un seul processeur, CPU0. L'appel sched_init() ne peut être déplacé avant l'appel smp_prepare_boot_cpu(), car l'initialisation de l'ordonnanceur dépend et exige que tous les processeurs soient déjà initialisés.

Ces mêmes limitations sont aussi applicables au traçage. Pour commencer le traçage très tôt, il faut placer l'appel d'initialisation de Ftrace *ftrace_init()* au début de la fonction start_kernel(). Comme tous les traceurs Linux (Lttng ou Perf), Ftrace utilise l'allocation dynamique pour réserver l'espace mémoire requis pour son tampon (ou *ringbuffer*). L'utilisation de l'allocation dynamique très tôt lors du démarrage n'est pas possible. En effet, on ne peut pas allouer dynamiquement de mémoire avant que le noyau initialise ses allocateurs de

mémoire : *kmalloc, vmalloc, slab* ou *page allocator*. La fonction mm_init() est responsable d'initialiser les allocateurs de mémoire, donc l'initialisation de Ftrace n'est pas possible avant mm_init().

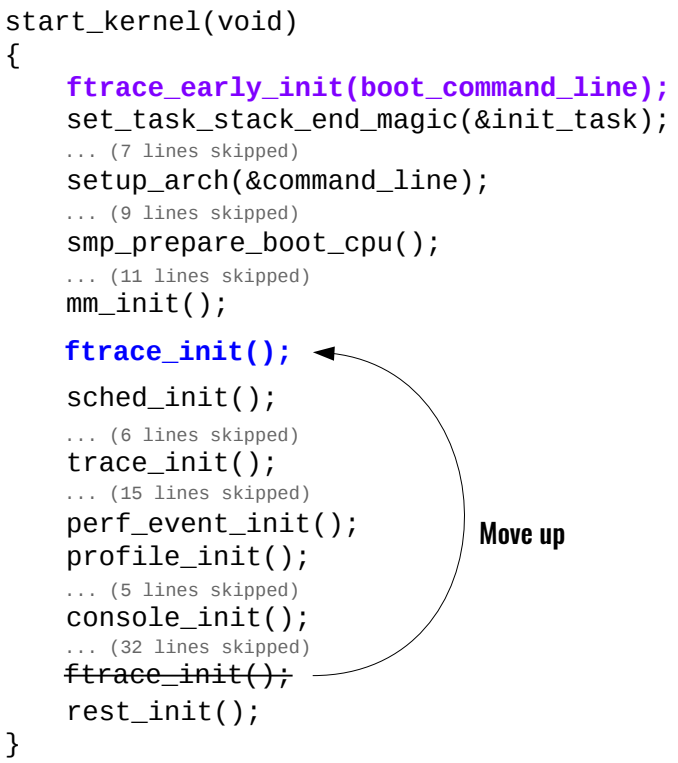

Figure 3.2 Activation du traçage tôt lors du démarrage

En discutant de ce problème avec l'auteur de Ftrace, Steven Rostedt, il a confirmé la difficulté de déplacer ftrace_init() plutôt que mm_init(), mais il était d'accord de faire les changements nécessaires pour au moins initialiser Ftrace immédiatement après l'initialisation de la mémoire mm_init(), tel que montré dans la figure 3.2. Jusque là, le traçage de fonction ne peut démarrer qu'après l'initialisation de la mémoire au démarrage. Pour pouvoir activer ce dernier, il faut ajouter "ftrace=function" à la ligne de commande du noyau, et le traçage commencera après la configuration de la mémoire.

Afin de tracer les fonctions avant cela, nous avons développé cette contribution qui permet de commencer le traçage dès le début de la fonction start_kernel(), (par l'appel ftrace_early_init() que nous avons ajouté), puis de l'arrêter lors de l'appel de ftrace_init(). Pour ce faire, nous avons modifié le code assembleur qui fait la gestion du *hook mcount* afin d'appeler notre fonction de rappel (*callback*) quand le mode *very_early* du traçage est activé.

Maintenant, à chaque fois qu'une fonction est appelée, notre *callback* sera appelé aussi. Nous

allons placer les évènements dans un tampon temporaire statique, qui sera copié dans le tampon de Ftrace après la configuration de la mémoire. La taille de ce tampon temporaire est définie par l'option VERY_EARLY_FTRACE_BUF_SHIFT qui peut être modifiée au moment de la compilation du noyau.

Le tampon temporaire utilisé est créé dans la section ".init" en utilisant la macro "___inidata". Le noyau Linux recyclera cette section après le démarrage, pour éviter de gaspiller cet espace une fois cette phase terminée. Ce mode de traçage n'est présentement disponible que sur les architectures x86 32-bits et 64-bits.

Hypergraph : Nous avons développé ce plug-in dans Ftrace, afin de tracer les entrées et sorties des fonctions du noyau invité. Cet outil nous permet de récupérer les piles d'appels d'un système invité durant les phases sensibles, difficiles a déboguer, comme le démarrage, la fermeture ou un plantage du noyau.

Une autre fonctionnalité très utile de cet outil est le fait d'associer les piles d'appels des processus du système invité avec les *vmexits* générés par ce dernier.

Après avoir installé hypergraph dans le système invité, nous avons exécuté la commande suivante :

```
guest:$ sudo trace-cmd record -p hypergraph
    plugin 'hypergraph'
  Hit Ctrl^C to stop recording
```

Pour collecter les évènements hypercall envoyé depuis le système invité, nous exécutons la commande suivant dans le système hôte, nous collectons aussi les évènements *vmexit* pour déterminer les types de sortie causés par le noyau invité :

```
host:$ sudo trace-cmd record -e kvm_hypercall -e kvm_exit -f exit_reason!=18
  /sys/kernel/debug/tracing/events/kvm/kvm_hypercall
  /sys/kernel/debug/tracing/events/kvm/kvm_exit
  Hit Ctrl^C to stop recording

0.677113: kvm_hypercall: nr 0x65 a0 0xffffffff96c85c90 a1 0x6e6 a2 0xb a3 0x0
0.677114: kvm_exit:      reason MSR_WRITE rip 0xffffffff96c5641d info 0 0
0.677124: kvm_exit:      reason MSR_WRITE rip 0xffffffff96c61eb4 info 0 0
0.677126: kvm_hypercall: nr 0x64 a0 0xffffffff974f53b0 a1 0xb a2 0x0 a3 0x0
...
```

Nous avons implémenté un parseur d'évènements afin de convertir les kvm_hypercall en évènements appropriés, la sortie sera générée comme suite :

```
CPU    depth      FUNCTION CALLS
 |       |        |   |   |   |
----------------------------------------
1)  kworker/0-114 => compiz-1661
----------------------------------------
1)     | 0  | process_one_work() \{
1)     | 1  |   blk_delay_work() \{
1)     | 2  |     __blk_run_queue() \{
...
1)     | 10 |                     smp_apic_timer_interrupt() {
1)     | 11 |                       irq_enter() {
1)     | 12 |                         rcu_irq_enter();
1)     | 11 |                       }
1) exit reason MSR_WRITE
1) exit reason MSR_WRITE
1)     | 11 |                       __do_softirq() {
1) exit reason PENDING_INTERRUPT
1) exit reason IO_INSTRUCTION
1) exit reason IO_INSTRUCTION
1)     | 11 |                       } /* __do_softirq */
1)     | 11 |                       idle_cpu();
1)     | 11 |                       rcu_irq_exit();
1)     | 10 |                     }
```

D'après les résultats obtenus, on découvre que le processus *compiz* n'a pas causé les sorties MSR_WRITE, PENDING_INTERRUPT et IO_INSTRUCTION. C'est plutôt la routine de gestion d'interruption *timer* et *softirq* qui les a causés.

Avec le hypergraph, on peut identifier avec précision quelle fonction dans le noyau invité est responsable des baisses de performance de virtualisation.

Bootlevel : Nous avons proposé cette contribution pour pouvoir tracer les niveaux d'initialisation du noyau Linux.

Le noyau Linux offre un mécanisme efficace, nommé *initcall*, pour déterminer l'ordre correct de l'initialisation des sous-systèmes et modules intégrés. Il existe actuellement 12 niveaux différents d'initialisation appelés dans l'ordre suivant :

```
security -> console -> early -> pure -> core -> postcore -> arch -> subsys --\
                                                                              |
                                          late <- device <- rootfs <- fs <-/
```

Nous avons utilisé l'outil hypergraph pour activer le traçage de fonctions, en combinaison

avec l'outil bootlevel. Nous pouvons activer/désactiver le traçage pour chaque niveau d'initialisation. Le tableau 3.1 présente quelques statistiques que nous avons obtenues en combinant les deux outils. D'après les résultats obtenus, on peut remarquer que le niveau *fs* (*filesystem*)

Tableau 3.1 Traçage d'appel de fonction pour chaque niveau d'initialisation. Le démarrage du noyau a duré pendant 2.7 seconds durant le traçages

Niveau de démarrage	Durée (ms)	Part du démarrage	Taille de trace
console	0.83	0.03 %	0.1 KB
security	138.96	4.8 %	2 KB
early	93.70	3.37 %	1.4 MB
pure	2.62	0.09 %	385 kB
core, core_sync	4	0.14 %	541 kB
postcore, postcore_sync	2.77	0.10 %	385 kB
arch, arch_sync	16.77	0.60 %	2.1 MB
subsys, subsys_sync	143.08	5.15 %	7.5 MB
fs, fs_sync	1871.23	67 %	155 MB
rootfs	6.05	0.2 %	721 kB
device, device_sync	410.7	15 %	26.4 MB
late, late_sync	90.2	3.25 %	16.6 MB

a occupé 67% du temps de démarrage du noyau, tandis qu'il existe d'autres niveaux qui ont pris seulement 1% de ce temps. Ce détail est important à connaître avant de commencer à tracer le démarrage, car l'analyse de trace sera encore plus facile si le traçage est activé intelligemment dans la zone investiguée.

3.3.2 Contribution à l'hyperviseur KVM

Nous avons proposé des modifications à l'hyperviseur KVM maintenu par Red Hat pour supporter l'activation rapide des hypercalls depuis des niveaux de machine virtuelle imbriquée. Cette contribution permet de désactiver la multiplication de sortie, cela empêche les instructions VMCALL exécutées depuis L2 (ou Ln, avec n > 1) d'être transmises à L1, car une seule sortie de L2 peut provoquer de nombreuses sorties L1. En effet, l'hyperviseur au niveau L1 exécutera des instructions privilégiées pour traiter les sorties transmises depuis L0.

Cette optimisation rend une seule sortie *vmcall* rapide et réduit la fréquence des sorties, tel qu'illustré à la figure 3.3.

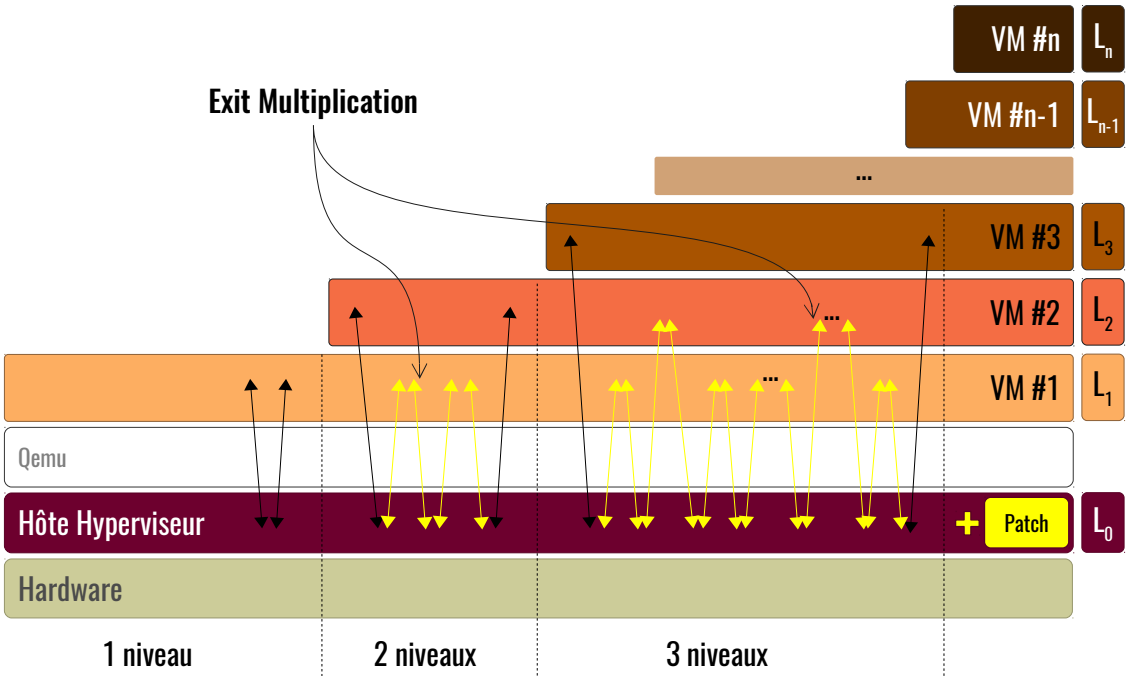

Figure 3.3 Désactivation de la multiplication de sortie

CHAPITRE 4 ARTICLE 1 : HYPERTRACING : TRACING THROUGH VIRTUALIZATION LAYERS

Authors

Abderrahmane Benbachir
École Polytechnique de Montréal
abderrahmane.benbachir@polymtl.ca

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

Submitted to Transactions on Cloud Computing

4.1 Abstract

Cloud computing enables on-demand access to remote computing resources. It provides dynamic scalability and elasticity with a low upfront cost. As the adoption of this computing model is rapidly growing, this increases the system complexity, since virtual machines (VMs) running on multiple virtualization layers become very difficult to monitor without interfering with their performance. In this paper, we present hypertracing, a novel method for tracing VMs by using various paravirtualization techniques, enabling efficient monitoring across virtualization boundaries. Hypertracing is a monitoring infrastructure that facilitates seamless trace sharing among host and guests. Our toolchain can detect latencies and their root causes within VMs, even for boot-up and shutdown sequences, whereas existing tools fail to handle these cases. We propose a new hypervisor optimization, for handling efficient nested paravirtualization, which allows hypertracing to be enabled in any nested environment without triggering VM exit multiplication. This is a significant improvement over current monitoring tools, with their large I/O overhead associated with activating monitoring within each virtualization layer.

4.2 Introduction

Cloud computing has emerged as another paradigm in which a user or an organization can dynamically rent storage resources and remote computing facilities. It allows application providers to assign resources on-request, and to adjust the quantity of resources to fit the workload. Cloud computing benefits lie in its Pay-as-you-Go model ; users simply pay for the used resources and can adaptively increment or decrement the capacity of the resources assigned to them [4].
In any case, an essential metric in this dynamic process is relative to the fact that, irrespective of cloud users being able to request more resources at any time, some delay may be needed for the procured VMs to become available. Cloud providers require time to select a suitable node for the VM instance in their data centers, for resources to be allocated (such as IP addresses) to the VM, as well as to copy or boot or even configure the entire OS image [37].
This dynamic process is unavoidable to ensure cloud elasticity. A long undesirable latency during this process may result in a degradation of elasticity responsiveness, which will immediately hurt the application performance.

Paravirtualization is the communication that happens between the hypervisor and the guest OS to enhance I/O efficiency and performance. This entails changing the OS kernel for the non-virtualizable instructions to be replaced with hypercalls that simply communicate with the virtualization layer hypervisor. The hypervisor, likewise, offers hypercall interfaces for related critical kernel activities like interrupt handling, timekeeping, and memory management [38].

An important challenge is to monitor guest systems while they go through critical stages such as boot-up and shutdown. Storage and network are not available at these stages. Memory allocation is not even possible in early boot-up, and memory corruption could also manifest during shutdown. Current monitoring tools were not conceived to operate in such conditions. To overcome those limitations, we propose paravirtualization solutions to be integrated into current monitoring tools.

In this paper, we propose **hypertracing**, a guest-host collaboration and communication design for monitoring purposes. In particular, we developed trace sharing techniques that enable accurate latency detection, which is not addressed by existing monitoring tools. We propose an approach based on different paravirtualization mechanisms, which is an effective and efficient way to communicate directly with the host, even from nested layers. The proposed method consists in offloading and merging guests and host traces. Depending on which com-

munication channels are used, trace fusion may need synchronization to insure that events are stored in chronological order.

Our main contributions in this paper are : **First**, we propose an hypercall interface as a new communication channel for trace sharing between host and guests, comparing this with shared-memory-based channels. The hypercall channel allows us to debug VM performance issues, even during sensitive phases such as boot-up and shutdown. **Secondly** we propose a technique to enable guest virtualization awareness, without accessing the host. **Thirdly** we submitted a kernel patch to the Linux community to enable function tracing during early boot-up. **Fourthly** we developed a KVM optimization patch for handling nested paravirtualization, which prevents exit multiplication and reduces CPU cache pollution. **Lastly**, we implemented VM analysis views which improve the overall performance investigation.

The rest of this paper is structured as follows : Section 4.3 presents a summary of existing paravirtualization approaches used for VM monitoring. Section 4.4 states the problem addressed by this paper. Section 4.6 introduces a comparative study of existing inter-VM communication channels. In section 4.7 we present our trace sharing design architecture tailored for monitoring purposes. Section 4.8 outlines some performance challenges of using hypertracing from nested environments and how we addressed them. Section 4.9 shows some representative use cases and their analysis results, followed by the overhead analysis in section 4.10. Finally, Section 4.11 concludes the paper with future directions.

4.3 Related Work

Various monitoring tools embraced paravirtualization to reduce I/O performance issues while tracing VMs. In this section, we summarize most of the previous studies related to our work, grouped into the following categories.

4.3.1 Hypercall based

Khen at al. [34] presents LgDb, a framework tool that uses Lguest to provide kernel development and testing, code profiling and code coverage. It enables running within a virtual environment kernel modules under study. The key behind LgDB is similar to traditional debuggers, as the framework will cause a context switch from the guest to the hypervisor mode, in order to stop guest execution when code flow reaches a specific point. Two approaches were proposed for this matter. First, the hypercall-based approach, which requires manual instru-

mentation of the inspected kernel code to set breakpoints. Secondly, the debugger-based approach which does not require any code modification, uses the kernel debugger (KGDB) to enable breakpoints over a virtio serial port.

Stolfa at al. [59] propose Dtrace-virt, an extension on top of the DTrace tracer in FreeBSD. It is a monitoring tool tailored for tracing untrusted VMs. DTrace-virt was designed as an Intrusion Detection System (IDS) and malware analysis tools ; this tool has been designed to detect malicious code injections by sending aggregated trace event data to the host hypervisor, using hypercalls as a fast trap mechanism, instead of using networking protocols (traditionally used by many distributed tracing tools) which induce a larger overhead in a virtualized environment. The authors did not provide any use case to evaluate the usefulness of such a monitoring approach for security purposes ; their work only discusses the infrastructure design and implementation challenges of their approach.

Gebai at al. [24] proposed a multi-level trace analysis which retrieves the preemption state for vCPUs by using merged traces from the host kernel and each VM. His work was extended by Biancheri [9] for nested VMs analysis. Both study the root cause of preemption by recovering the execution flow of a specific process, whether it is a host thread, VM or nested VM. They proposed an approach to resolve clock drift issues using paravirtualization with periodic hypercalls as synchronization events. Their approach adds about a 7% constant overhead to each VM, even if they are idle.

4.3.2 Page sharing based

Yunomae [61] implemented virtio-trace, a low-overhead monitoring system for collecting guests kernel trace events directly from the host side over virtio-serial, in order to prevent using the network stack layers while sending traces to host. They enable an agent inside the guest to perform splicing operations on Ftrace [53] ring buffers, when tracing is enabled. This mechanism provides efficient data transfers, without copying memory to the QEMU virtio-ring, which is then consumed directly by the host.

Jin at al. [32], developed a paravirtualized tool named Xenrelay. This tool is an unified one-way inter-VM transmission engine mechanism, very efficient for transferring frequently small trace data from the guest domain to the privileged domain. Xenrelay has the ability to relay data without lock or notification. This mechanism was implemented using a producer-consumer circular buffer with a mapping mechanism to avoid using synchronization. It helped to optimize Xen virtualization performance by improving the tracing and analysis virtualization issues.

4.3.3 Hypervisor monitoring

Nemati [43, 44] proposed a new approach to trace guests virtual CPU states, in any virtualization layer, tracing only the host without accessing VMs. Sharma [58], used a hardware tracing monitoring-based approach to generate hypervisor metrics and exposing virtualization overhead. This method involves continuous host access, which may not be possible in some cases. Our work is unique in the sense that it can be used without the need to access the host. We use paravirtualization to enable effective host-guest communication and collaboration. This enables virtualization-awareness from the guest perspective, and enables measuring the virtualization overhead from the guest.

Other studies [31, 12] attempt to measure the virtualization cost using performance benchmarks suites, in order to measure CPU, memory and I/O virtualization overhead. However, such an approach is not representative of real-world workloads.

Paravirtualization approaches are mostly used to avoid the heavy network path communication, using shared memory or hypercalls as fast communication paths in virtualized environments. To our knowledge, no previous study has used this mechanism to solve real world performance issues, for instance at bootup or shutdown when few operating system facilities are available ; paravirtualization approaches were mostly used to speedup I/O operations.

4.4 Problem Statement and Definitions

In this section, we present challenging issues, grouped into the following categories.

4.4.1 I/O Bottlenecks

Various performance issues in virtual environments are caused by the isolation layer imposed by the hypervisor. VMs have the illusion of exclusive access to system hardware due to the virtualization layer. When a system has multiple layers of virtualization, it becomes very difficult to know what is happening. As a basic approach, we can enable monitoring in each layer. However, there is still the challenge to merge and synchronize the traces coming from multiple layers due to clock drift. Moreover, monitoring in all layers comes with overhead concerns, as most monitoring tools need to consume the recorded data in one way or another, either by storing them to disk (offline analysis), or using live monitoring to offload them through a network (online analysis). Both approaches introduce significant overhead related to I/O operations, especially when tracing is enabled within a nested environment. In a nested

environment, the performance slowdown could reach a factor of 30 and more.

4.4.2 System Failures

QEMU has a panic device, supported by Libvirt since the first version. Using this device, the guest is capable of notifying the host when a guest panic occurs, through the process of sending a particular event directly to QEMU, which in turn will inform Libvirt.

Libvirt offers a mechanism that can store guest crash dumps automatically to a dedicated address on a host. Activating the panic device PVPanic for guests will enable a crash dump to be forwarded, without having kdump [27] running within the guest [51]. The major issue with this method is the amount of data generated. A basic crash dump size would be 128MB. If a system has 1TB of memory, then 192MB will be reserved, a 128MB of basic memory plus an additional 64MB. As a result, the host would need more storage resources to store all these crash dumps coming from different guests.

4.4.3 Boot-up

The Boot-up time is the amount of time it takes to boot a VM into a ready state. It is an important factor for VM allocation strategies, particularly while provisioning for peak load. Automated allocation strategies will request new VMs to match load demand. In this situation, the time to boot VMs is critical, as the system is waiting after the boot-up in order to be fully operational. Enabling monitoring during the boot-up phase is challenging, yet is often the only effective way to investigate boot-up latencies. Data generated during this phase cannot be stored on disk or offloaded through the network, because network and storage are only available once this phase is almost completed.

Another approach may be possible. If enough memory space is allocated at the beginning of boot-up, to prevent event loss, then events can be consumed after boot-up when storage and network are available. However, this approach requires a large memory space allocation (ring buffer). It also adds a significant time overhead because the monitoring tool uses memory frame allocation at initialization time, avoiding postponed page faults while tracing is enabled. This initialization, at the start of the boot process, directly impacts the boot time.

Kernel crashes may also happen while VMs are booting. Investigating such problems is a non-trivial task. Kernel developers only use serial consoles for dumping messages such as call stacks and register values, which is not enough in most cases to solve these issues.

4.4.4 Shutdown

Investigating the shutdown latency is another challenging task. One may question the interest of optimizing the shutdown latency. In many cases, latencies are not very important when turning off VMs during the under-provisioning stage. Nevertheless, VMs shutting down consume resources and may prevent a host from starting up more rapidly newer VMs. Shutting down existing VMs quickly is the fastest way to get back the necessary resources. Thus, any delay during the shutdown is directly impacting the provisioning of newer VMs. However, in some cases, the shutdown latency is much more critical. Indeed, in critical embedded systems, the shutdown latency is on the critical path for the case where your system needs to reboot after doing a major software upgrade. Any significant delay impacts the non availability phase during the upgrade.

When a system is shutting down, userland processes are being released, drivers being unloaded, and the network and disks are not available. Current monitoring tools are not designed to operate during this phase.

In this paper, we present how we use paravirtualization to design guest-host collaboration and address the above challenges. As we illustrate in section 'Use cases', the techniques resulting from our work will help to efficiently investigate system failures and hidden latencies within VMs, as well as enabling runtime perturbations detection due to CPU or disk sharing between colocated VMs.

4.5 Environment

For all experiments and benchmarks, we used a computer with a quad-core Intel(R) Core(TM) i7-6700K CPU running at 4.00GHz, 32 GB of DDR3 memory and a 256 GB SSD. The operating system is Ubuntu 16.10, running Linux 4.14 and LTTng 2.10. The frequency scaling governor of the CPU was always set at "performance" to prevent CPU scaling operations while performing the benchmarks. Benchmark results were computed using the average of one million iterations.

KVM and QEMU 2.8.0 were used as virtualization technology. Host and guests were configured with the same linux versions during all experiments. For nested environments, we enabled all nested optimizations such as the EPT-on-EPT feature.

4.6 Inter-VM Communication Channels Analysis

Virtualization technology provides security and isolation, with the ability to share system resources between VMs co-located on the same host. Furthermore, isolation is the main property that keeps the industry from shifting to container technology. However, when co-located VMs need to communicate with each other or with the host, it becomes a bottleneck. Networking is the standard communication channel which has mostly been used in the cloud system.
Communication by means of TCP/IP requires a longer time since the transfer of data from the sender VM directly to the receiver host undergoes a long communication channel through a host hypervisor that incurs multiple switches between the root mode and user mode, as well as going across the entire network stack layers. This is inherently inefficient and causes performance degradation. Moreover, to allow efficient and fast communication, the guest OS and the hypervisor are capable of using an optimized I/O interface, rather than depending on TCP/IP. This mechanism that is called **paravirtualization** [50].

In this section, we discuss different paravirtualization communication channel alternatives to TCP/IP, exposing design limitations and performance drawbacks of each one.

4.6.1 Page Sharing Channel

Instead of emulating hardware devices, Xen designed a new split driver I/O model, which provides efficient I/O virtualization and offers protection and isolation of the device interface. In this new model, drivers are split into frontend and backend, isolated from each other across Xen domains. These frontend and backend can only communicate asynchronously through the hypervisor, in order to satisfy the isolation, using shared-memory (shared I/O ring buffers) [3].

The shared I/O ring buffers are enabled by two main communication channels provided by Xen, the Grant Table and the Event mechanism. The event mechanism allow inter-VM asynchronous notification and signals, instead of polling. Events can be used to signal received/pending network data, or to indicate the completion of a disk request. The Grant Table is the fundamental core component for memory sharing. It offers mapping/transfer memory pages between frontend and backend drivers. Moreover, the grant mechanism provides an API, accessible directly from the OS kernel, to share guest pages efficiently and securely. It allows guest domains to revoke page access at any time.

The Xen page sharing mechanism is implemented by issuing synchronous calls to the hypervisor (hypercalls). If a guest domain wants to share a page, it invokes a hypercall that

contains the target domain reference and the shared page address. The hypervisor will first validate the page ownership, then map the page into the address space of the target domain. Similarly, to revoke a shared page, the guest issues another hypercall, and this time the hypervisor unmaps and unpins the guest page from the target address space [49].

In early 2006, when KVM was merged into linux, virtio was introduced. It was based on Xen I/O paravirtualization, which uses a mechanism similar to the Grant Table. Afterwards, virtio evolved and converged into the default I/O infrastructure supported by different hypervisors.

Virtio is an efficient paravirtualization framework that provides a transport abstraction layer over hardware devices. It offers a common API to reduce code duplication in virtual device drivers. Such drivers enable efficient transport mechanisms for guest-host as well as inter-guest communication channels [56]. Similar to Xen, virtio device drivers are also split into the frontend which resides in the guest OS, while the backend can be either in user space, inside QEMU, or in kernel space as KVM modules (vhost). The communication between those drivers (guest and hypervisor) is organized as two separate circular buffers (virtqueue), in a producer-consumer fashion, as illustrated in Fig. 4.1. The avail-ring is for sending and the used-ring for receiving.

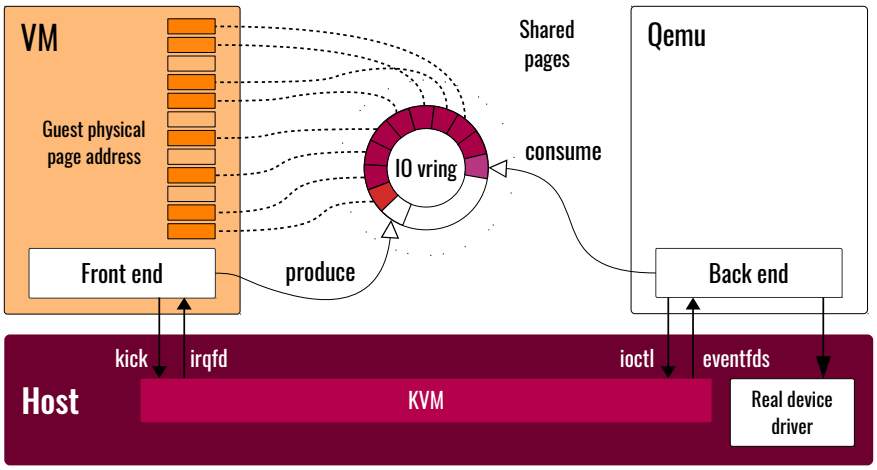

Figure 4.1 Architecture of Page Sharing mechanism in virtio

A producer (guest driver) writes data into the virtqueue avail-ring in the form of scatter-gather (guest physical address pointer and length) lists, then notifies the host when buffers are available, using the kick call mechanism. The virtio kick operation can be performed using the *paravirt_ops* API infrastructure. This API can be either an input/output port access (*pio*) operation, a hypercall or other notification mechanism which depends on the hypervisor. The host can decide to disable guest notifications while processing buffers, as notification implies an expensive context switch of the guest. Virtio also supports batching,

which improves the performance throughout. This feature can be enabled by adding multiple buffers to the virtqueue before calling the kick event [35]. When the consumer (hypervisor or QEMU) gets the notification, it reads the virtqueue avail-ring buffers. The consumed buffers will then be pushed to the used-ring.

A virtio device may choose to define as many virtqueues as needed. For instance, the virtio-net device has two virtqueues, the first one for packet transmission and the other one for packet reception.

Virtio-serial is an implementation of the serial bus on QEMU. This mechanism was used for monitoring purposes, such as retrieving metrics of the guest CPU, memory and disk usage [46]. As presented in section 4.3, virtio-serial was also used as a transport layer to implement virtio-trace. We decided to use virtio-serial as the reference implementation for the page sharing channel, when we compare it to other inter-VM communication mechanisms later in this section.

4.6.2 Memory Sharing Region Channel

Inter-Process Communication (IPC) is an operating system mechanism allowing different processes to collaborate and share data. A variety of IPC mechanisms are supported in most operating systems, including sockets, signals, pipes and shared memory. Depending on the system configuration, processes may be distributed across different machines, or across different data centers, which requires using remote procedure calls (RPCs) and streaming data over the network. On the other hand, processes may be located on the same machine and, in this case, it is more efficient to use a shared memory to increase the performance of concurrent processing.

With the recent advances in virtualization technology, the need for efficient Inter-VM communication is growing. A variety of inter-VM shared memory mechanisms were introduced in the Xen hypervisor using a page-flipping (Grant) mechanism. Nahanni and ZIVM are two shared memory mechanisms that were introduced in KVM. Unlike Xen, these mechanisms allow host-guest as well as guest-guest sharing, supporting POSIX and System V shared memory regions.

Developed by Cameron Macdonell, Nahanni is a zero-copy low-latency and high-bandwidth POSIX shared memory based communication channel. It enables hosts, guests, as well as inter-guests efficient transport mechanisms. Unlike a page sharing mechanism, it does not require any hypervisor or guest involvement while reading or writing to the shared region [36].

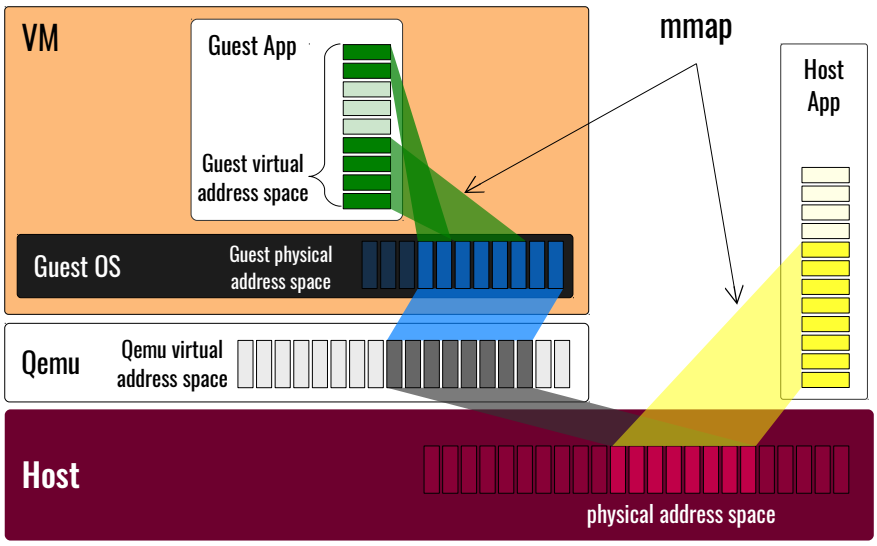

Figure 4.2 Architecture of Memory Sharing Region

As illustrated in Fig. 4.2, Nahanni is implemented as a new virtual PCI device (ivshmem) in QEMU, which will be used to share the POSIX memory region, mapped into QEMU with the guest OS. Userspace applications inside the guest OS can access directly the shared memory region by opening the device using the *mmap* system call. This call will map the memory region into their address space, with zero-copy. Nahanni also supports a notification mechanism, in order to notify the host or other guests about data availability. It uses event file descriptors (eventfds) as an interrupt for signalling and synchronization mechanisms.

We decided to use Nahanni as the reference implementation for the memory sharing channel, after we compared it with other inter-VM communication mechanisms.

4.6.3 Hypercall Channel

When the guest OS is running in user mode, all executed sensitive instructions should cause a trap into the hypervisor, incurring significant overhead. To address that, using paravirtualization, a guest OS source code will be modified to replace those sensitive instructions with a single invocation of the hypervisor to perform heavy-weight work using hypercalls. The hypercall interface allows the request of a service from the hypervisor by performing a synchronous software trap into the host, which is similar to system calls. Fig. 4.3 illustrates the hypercall trap mechanism in more details.

In other words, the hypercall is a guest request to the host to perform a privileged operation, and it can be used as a guest to host communication mechanism as well.

The hypercall transport mechanism is performed using registers. As an illustration for x86 architecture, guests could use the default general purpose registers **ax**, **bx**, **cx**, **dx** and **si** to place data before emitting the hypercall. The ax register will be reserved for the hypercall number during the submission, then the hypervisor will use **ax** to place the returned value when switching back to user mode [48]. On the Intel 64-bits architecture, additional registers can be used such as **r8** to **r15**, coupled with a proper compiler register clobbering mechanism.

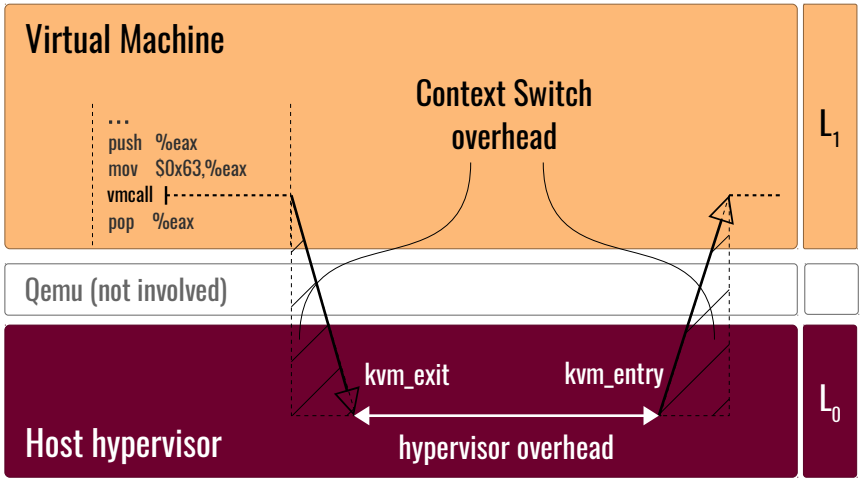

Figure 4.3 Hypercall in a nutshell

A practical hypercall use case is available in the Linux project. A hypercall can be used by guests to wake up a virtual CPU (vCPU) from the halt (HLT) state. In case a given thread is waiting to access a resource, the thread would perform an active polling (spinlock) on a vCPU. Paravirtualization can be used in this case to save wasted cpu cycles, by executing the HLT instruction when a time interval threshold was elapsed. The execution of the HLT instruction causes a trap to the hypervisor, and then the vCPU will be put into sleep mode until an appropriate event is raised. When the resource is released by other threads, on other vCPUs, the thread can wake up the sleeping vCPU by the triggering KVM_HC_KICK_CPU hypercall [48].

4.6.4 Performance Comparisons

In this section, we present the performance comparison of three data transmission channels : hypercall, virtio-serial and Nahanni. We measure the bandwidth for streaming a fixed volume of data (1 GB), from guest (with single vCPU) to host repeatedly. Our bandwidth benchmark measures the time to send the data to the host, without measuring the overhead of consuming it. There are several ways for consuming the data, and in this section we only need to compare

the transmission path between these channels, which is enough for a micro-benchmark. Later in section 4.10 we will compare these channels with a more realistic workload.

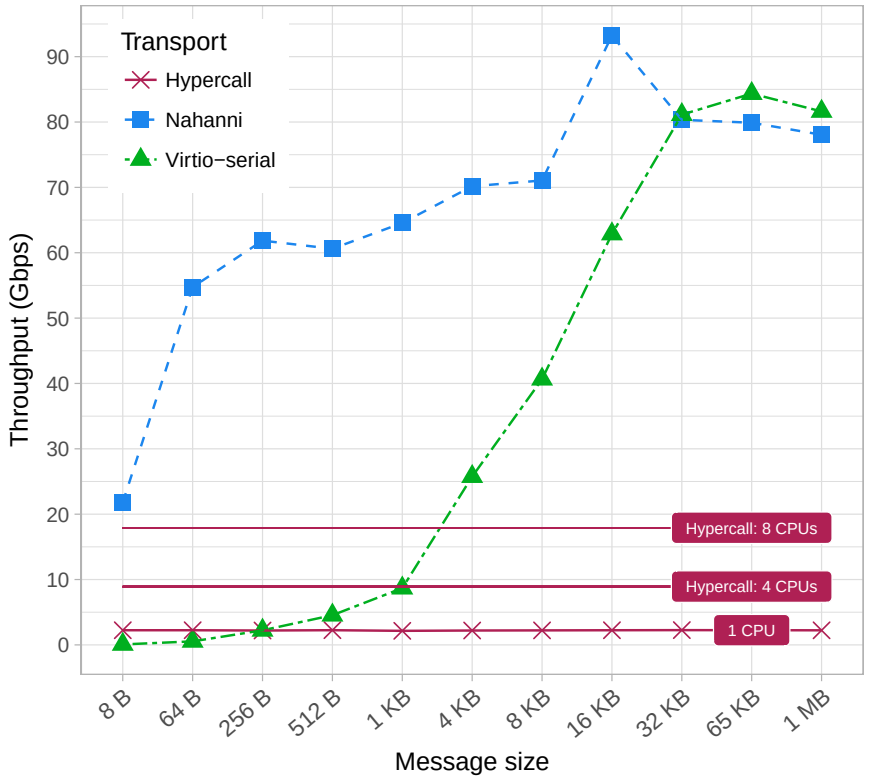

Figure 4.4 Throughput versus message size for Inter-VM communication transport benchmarks

Fig. 4.4 shows the bandwidth measured with hypercall, virtio-serial and Nahanni while increasing the sending message size. The bandwidth increases for both virtio and Nahanni with larger message sizes. Then, both reach a stable throughput of 80 Gbps when message sizes are larger than 32 KB. Nahanni achieves higher throughput for smaller and larger message data than both the hypercall and virtio mechanisms. Virtio shows the worst bandwidth for sending smaller messages. This is because it requires a larger number of hypervisor exits to send the same amount of bytes.
The hypercall achieves a constant throughput, of about 2 Gbps, with a single CPU, and 17.5 Gbps with 8 CPUs. This is because the hypercall uses CPU registers for the transmission. Increasing the message size simply implies using more registers (if available). We couldn't use more than 8 CPUs due to resource limitations.

In summary, shared memory based approaches achieved a higher communication efficiency as compared to other channels.
To solve the challenging problems previously raised in section 4.4, we focus in this paper

on two communication channels : hypercall and shared memory region. The next section presents the design of hypertracing techniques using both channels.

4.7 Trace Sharing Design and Architecture

In this section, we will explain the design principles of the two types of Inter-VM communication channels : hypercall and memory sharing channels.

4.7.1 Hypercall

As explained in section 4.6, a hypercall is a guest synchronous request to the host to perform a privileged operation, but it can also be used for debugging, or for guest-host communication purposes. The heart of a hypercall is the *vmcall* VMX instruction, the main aim of which is to cause an instant trap into the host hypervisor. On the other hand, the transport mechanism of a hypercall can be performed solely by using guest registers. The approach of using the *vmcall* instruction alongside the registers requires saving these registers into the guest stack. When returning from the hypercall, the guest can then restore the registers state, from the stack, before continuing execution. The *vmcall* instruction can also be used without touching guest registers. Instead, it can be used for the purpose of inspecting guest registers, for instance as a breakpoint mechanism.

Debugging aspect

The *vmcall* instruction has a behavior similar to debugger breakpoint mechanisms, which use the interrupt instruction (*INT 3*) to cause a userspace program to trap into the kernel space.
A practical example of debugging a guest kernel using *vmcall*, would be the function tracing. Adding the gcc *-pg* option while compiling the kernel will automatically add a mcount() call at the beginning of each function. Ftrace [53], the kernel tracer, relies on the gcc *-pg* option for its function tracing infrastructure. During boot-up, Ftrace will use live patching to convert all mcount() calls into *NOP* instructions. When tracing is enabled, Ftrace converts the *NOP* instructions into a callback routine that handles the tracing part. This mechanism is known as dynamic tracing.

To be able to debug a guest kernel from the host side, we can use dynamic tracing coupled with *vmcall*. This mechanism will convert dynamically the *NOP* instructions into a *vmcall* instruction. Each time a kernel function is called in the guest, a trap to the host hypervisor

happens. Once in the host, it is possible to inspect the guest registers, stack, heap and memory directly from the host. This allows the extraction of guest data such as function address or function arguments. It is possible to use a pause/resume operation on a guest, when a specific hypercall was triggered. This feature will allow developers to manually inspect guests for advanced step-by-step investigations.

Tracing aspect

Debugging is a very useful technique, providing developers with a handful of features to investigate very complex problems in step-by-step mode. However, debugging in production cloud-based environments is not tolerable due to its huge overhead cost. Instead, practitioners use tracing tools to collect VM execution events in the most optimal way. Then, collected events would be asynchronously recorded on disk for offline analysis.
Another way of tracing virtual environments would be to offload guest events directly to the host. This mechanism requires using a transport channel, such as hypercall. When offloading guest events using hypercalls, there are three different offloading configurations to consider, discussed as follows :

One-to-One : 1-event to 1-hypercall This configuration restricts the data transmission to only one hypercall per event. Each guest event will cause a trap in the host hypervisor, using host timestamps when recording the guest event.

In the case of high event frequency, this configuration may dramatically impact the overall performance of the guest execution. However, no latency is added to traced events and, as a result, we have instant insight into what is happening inside guests, from a host perspective with aligned timing. This mechanism allows tracing many guests at the same time in live monitoring.

One-to-Many : 1-event to n-hypercalls This configuration may help the situation when a specific event needs more memory space than available (e.g. fixed number of registers) to send a payload. In this case, two or three hypercalls may be used for the transmission, resulting in an enormous performance impact on the guest execution.
In this paper, we didn't find a need for this configuration. First, all traced event data payloads did fit between 1 to 4 registers. Secondly, a shared page can be shared between the host and guest in order to expand the hypercall payload maximum size.

Many-to-One : n-events to 1-hypercall The purpose of this configuration is mainly to reduce the overhead induced by a hypercall. The idea is simple, multiple hypercalls can be batched as a single hypercall. This is important to reduce the number of hypercalls being used for sending events. The more hypercalls are batched, the less overhead we get. We named this mechanism the **event batching** and **event compression**.
When event batching is enabled, we save the payload and record the guest timestamp on each event. When the last event of the batch is encountered, we group the events data as much as possible to fit into the hypercall payload. Instead of sending a guest timestamp, we send the time delta of each event, which would be computed against the guest last event. The last event will trigger a hypercall and a timestamp will then be recorded in the host, which will be used to convert the batched events times delta into host timestamps. Storing the time delta requires only 32 bits, and enables batching events within a maximum interval of 4 seconds. This is sufficient for the groupings envisioned.

In order to measure the effectiveness of the batching mechanism, we study the cost of tracing guest kernel events using the traditional tracers such as Lttng, Ftrace and Perf. Then, we compare those tracers against hypertracing with batching enabled.

From the results, presented in Fig. 4.5, we find that batching reduced greatly the hypertracing overhead compared with other tracers. We notice that the overhead is reduced by a factor of 3 (from 425 ns to 150 ns) with minimal batching (2 events). This performance is even higher than that of the Perf tracer. The batching cost is a constant overhead of about 25 ns. It involves recording guest timestamps on each event, and computing each event (grouped) time delta to be stored in the payload.
Hypertracing outperforms the LTTng and Ftrace tracers when batching 5 events or more together. Thus, with a relatively small number of batched events, it performs better than the finest tracers.
Batching many events may result in having less space for the payload of each event within the hypercall. Batching 15 events results in a 50 ns overhead per event, but in return the hypercall payload will only have 24 bytes available. Therefore, batching 9 events may be better than batching 15 events, because it doubles the payload length to 48 bytes, in exchange for just an additional 12 ns overhead.

Tracing through hypercalls involves overhead in different layers. This mechanism starts by hooking a probe callback in the guest trace event infrastructure. The registered callback

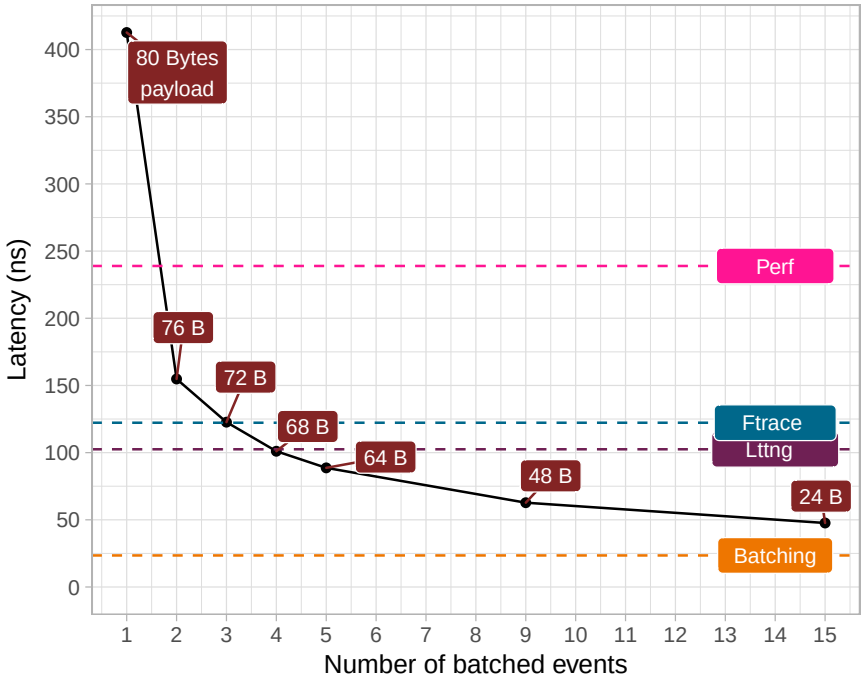

Figure 4.5 Offloading latency of the hypercall channel when using event batching.

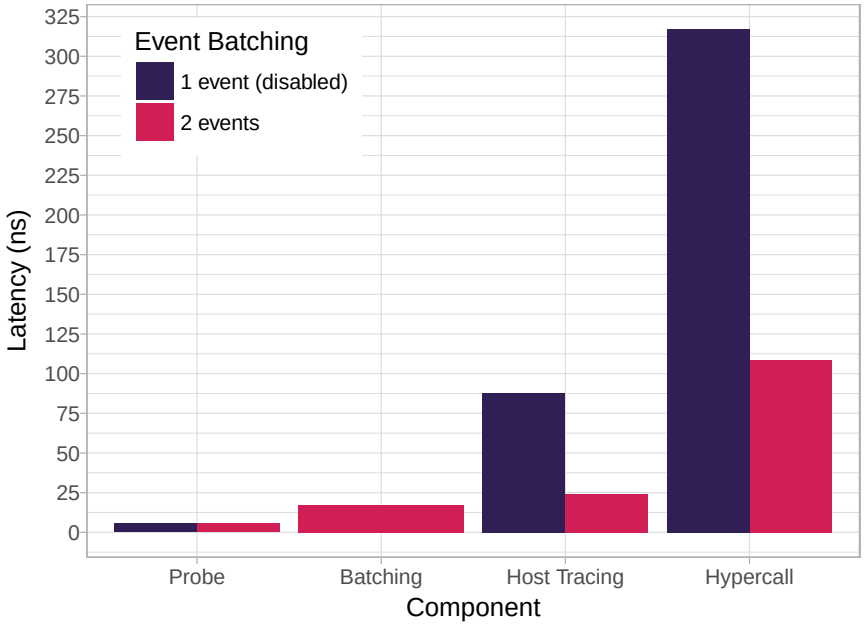

Figure 4.6 Different component overhead of the hypercall channel when using event batching.

would be called whenever these events occur. At this point, a hypercall would be triggered using one of the three offloading configurations previously explained. From the host side, the triggered hypercalls are gathered by using a tracer. Any tracer could be used to trace hypercalls, and in this work we are using LTTng due to its low tracing overhead.

In order to understand which part of tracing does impact the performance, we perform another experiment to measure the cost of each component. The results are shown in Fig. 4.6. By analyzing the results, we find that "batching" not only reduces the hypercall overhead but it does reduce the host tracing overhead as well. As can be seen in the figure, the hypercall is the main source of overhead, being 70% of the overall cost. Host tracing comes second with a 25% cost. On the other hand, using a minimal 2-events batching configuration adds 17 ns of overhead. In return, it reduces the hypercall and host tracing cost by 65% and 72% respectively.

Even with a minimal configuration, event batching turns out to be very efficient. However, this mechanism would mostly work for periodic events that happen frequently. Moreover, finding the appropriate number of events to batch largely depends on the type of events being traced. The main drawback of batching is the latency added before an event becomes visible, for instance in live tracing. Another problematic scenario is upon a crash, the crucial last few events before the crash may be held up in a batch being assembled.

In order to fit as many events as possible in the payload, the batched events should be periodic, frequent, and from the same event type. These properties are already present in most operating system applications. Nonetheless, in this paper we focus on the Linux operating system.
Linux offers a robust tracing infrastructure that supports static and dynamic instrumentation. Static tracepoints have been manually inserted into different subsystems, allowing developers to better understand the kernel runtime issues. Scheduling, memory management, filesystems, device drivers, system calls and many more are important components in the kernel, and they are exercised frequently during the execution. These components are already instrumented, and they can efficiently be used with event batching since they occur periodically in the system. As an illustration, Fig. 4.7 provides more details on how batching scheduling events could be implemented. We have used the sched_switch event as an example in this figure.

Sched_switch is the linux trace event that shows the context switches between tasks. This event allows recording information such as the name, priority and tid of the previous and the next tasks. This is represented by seven fields as follows : prev_prio, prev_state, prev_tid, prev_comm, next_prio, next_tid and next_comm. In linux, the pid/tid maximum value is configurable, but pid has a default maximum value of 32768, which needs only 15 bits to be

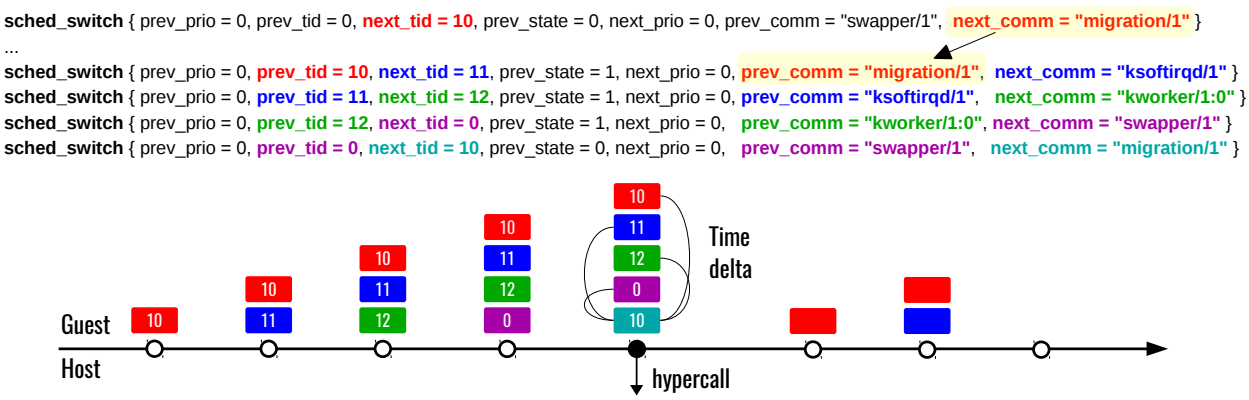

Figure 4.7 Event batching of schedule switches events with compression enabled.

stored. The task priority and it state can be stored within 8 bits each. The task name length is 16 characters, and requires two registers to be stored.
If we use a one-to-one configuration, each sched_switch event would require 5 registers (64 bits architecture), which can easily fit into a hypercall payload. Batching sched_switch events is much easier than it looks. As has been noted previously, the sched_switch event contains data fields about the previous and next tasks. We notice that, when analyzing two consecutive sched_switch events, as shown in Fig. 4.7, the task "migration/1" which was included as the "next task" in the first sched_switch event, is also included in the second sched_switch event as the "previous task". Therefore, **event compression** can be enabled, omitting the repeated information, in order to batch more sched_switch events within the payload.

The effectiveness of any compression algorithm is evaluated by how much the input size is reduced, which depends on the input data. In our case, the situation is different. We have a predefined and fixed length of hypercall payload, and our goal is to fit in that payload as many events as possible, without losing any information.

With compression enabled, event batching will take into consideration only the "next task" fields, as the "previous task" fields can be retrieved from the previous event while consuming these events from the host.
Using the above compression optimization will allow us to compress 3 sched_switch events while using only 9 registers : 1 register to store the time delta and 8 to store the event fields. To be able to compress more than 3 events, using a hash would be optimal for storing characters like task name which can be to fit into 32 bits, or use a guest-host shared page to store these characters as debug symbols.

A direct limitation of the event batching mechanism is that we often want to group together the function entry and exit while enabling function tracing, in order to reduce the overhead by 50%. However, functions that last a long time (e.g. close to the root of the call tree such as the main function) would not be collected if tracing was stopped before recording their exit event.

Performance analysis

The hypercall is a great technique for debugging, but it comes with a cost. We used a benchmark to evaluate the overhead of using hypercalls. In this evaluation, we are mostly interested in studying the evolution of consecutive hypercalls overhead over time. We used a VM with one CPU and pinned its vCPU0 to physical CPU0, and used a tight loop evaluation for this CPU intensive benchmark. We ran twice the benchmark, the first experiment was with CPU frequency scaling disabled, the other one enabled CPU scaling. The results obtained are presented in Fig. 4.8.

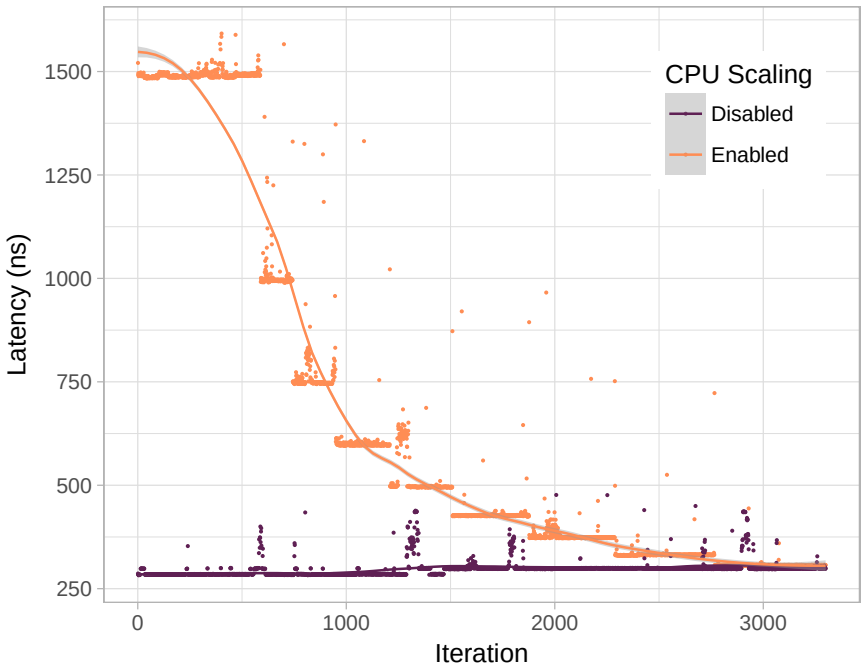

Figure 4.8 Consecutive hypercalls evolving overhead (for 2 secs).

Fig. 4.8 shows that, when CPU scaling is disabled, the measured hypercall overhead is stable between 280 and 300 ns. This CPU mode is mostly used during benchmarks because it reduces unreliable measurements related to the CPU scaling operation. When CPU scaling is enabled, we can see that it took 2 seconds for an idle CPU to get to the same performance level as

when CPU scaling was disabled. In other words, hypercall is a CPU bound mechanism. One should be careful not to overwhelm the system when running other CPU intensive workloads.

4.7.2 Memory Sharing

Shared-memory based communication channels are known for their high-bandwidth and performance, compared to other existing communication mechanisms. It allows efficient data sharing across virtualization boundaries, by enabling guest-host and inter-guests communication. When communicating with memory, the host hypervisor intervention is only required when performing notification or synchronization. Our primary aim is to prevent any host involvement while designing the trace sharing mechanism. For this reason, we do not use any notification mechanism.
In this paper, we mainly focus on communication between guest and host. It is an asynchronous-based communication, where the shared data is consumed without the need for explicit synchronization.

We developed the shared memory buffer as a virtual CPU producer-consumer circular buffer. In any buffering design scheme, data overflow may occur when the producer is writing faster than the consumer client reads. In this case, two modes may be used. The first one, is the blocking mode, where the producer suspends writing into the buffer when full, causing new data to be lost. The second, is the flight recorder mode, which overwrites older data in order to push recent data into the buffer.
The flight recorder mode is used here, continuously writing into the shared buffer, to prioritize the recent data. This mode constantly has the latest data, which is convenient when we want to take snapshots of the shared buffer on specific events such as guest exits, task migrations and more.

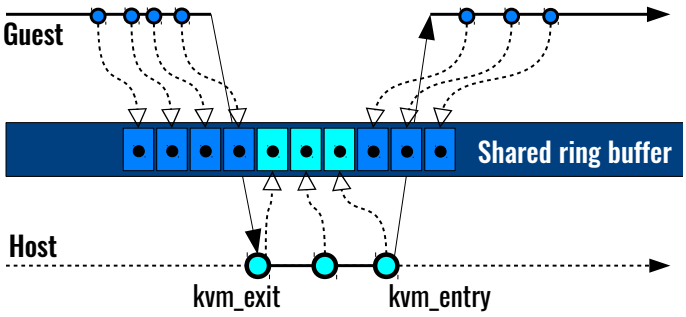

Figure 4.9 Shared ring buffer between guest and host.

Fig. 4.9 shows our shared buffer design. Both the guest and host do write into the shared

ring buffer but not at the same time. Guests may use any traditional monitoring tool for recording trace events into the shared buffer. When a trap happens (the hypervisor emulates guest privileged instructions), the host would then record guest related events (VMCS data) into the shared buffer of the preempted virtual CPU. In other words, the host would be responsible for offloading virtualization events to its guests for monitoring purposes, which would then enable guests to be aware of the virtualization operations performed by the hypervisor.

4.8 Nested Para-virtualization

Nested virtualization is the concept of enabling a VM to run as a hypervisor. VMs running inside guest hypervisors are called nested VMs. Once this feature is enabled by the Infrastructure-as-a-Service (IaaS) provider, cloud users will have the ability to manage and run their favorite hypervisor of choice (like Xen, VMware vSphere ESXi or KVM) as a VM.

The x86 virtualisation is a single-level architecture, it follows the "trap and emulate" model and supports only a single hypervisor mode. Running privileged instructions from a guest level or nested guest level should cause a trap to the host hypervisor (L_0). Any trap received by the host should be inspected ; if the trap was coming from a nested level, it should be forwarded to the above hypervisors for emulation. Since the guest hypervisor (L_1) is unmodified, it has the illusion of absolute control of the physical CPU. When L_1 receives the trap forwarded from L_0, L_1 will emulate the nested VM (L_2) trap using VMX instructions. VMX instructions are privileged instructions too, and they can only execute in root mode. In this case, VMX instructions should be emulated by L_0, which causes additional traps [6].

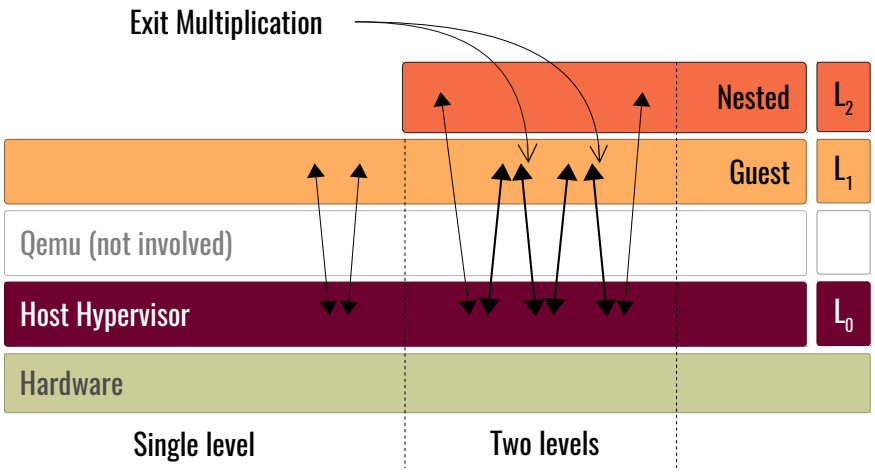

Figure 4.10 Exit multiplication, caused by nested traps

This phenomenon is called **exit multiplication**, which is illustrated in Fig. 4.10, where

a single high level L_2 exit causes many L_1 exits leading to performance degradation. The Turtles Project [6] proposed two optimizations implemented in KVM, in order to reduce L_1 exits frequency. The first one optimizes the transitions between L_1 and L_2 by optimizing the merging process of $VMCS_{0\to1}$ and $VMCS_{1\to2}$ into $VMCS_{0\to2}$, copying only the modified fields, and performing multiple fields copy at once. The second optimization relies on improving exit handling in L_1 by replacing *vmread* and *vmwrite* instructions by accessing directly $VMCS_{1\to2}$ by non-trapping memory *load* and *store* operations.

Nested paravirtualization is a way of performing paravirtualization from nested levels, which may result in exit multiplication. If the paravirtualization technique used relies heavily on the hypervisor, it would decrease the guest performance. As presented in section 4.6, Nahanni (memory sharing region) is the only mechanism that does not require any hypervisor involvement while reading and writing to the shared buffer. This mechanism enables efficient hypertracing through shared memory, where any nested guest may communicate with the host, guest or any nested guest without any performance degradation.

The virtio (page sharing) channel requires notification operations to perform inter-VM communication. This notification is a kick call performed by *pio* operation or a hypercall, either of which will lead to performance slowdown.

Using hypertracing through a hypercall channel may be the most costly mechanism to use from a nested level, because it purely relies on *vmcall*, which is a VMX instruction. To understand the cost involved in nested hypertracing, we performed micro-benchmarks of hypercalls from different layers L_1, L_2 and L_3. The results are presented in Table 4.1. The table shows that performing a single hypercall from L_1 costs only 286 nanoseconds, whereas performing a hypercall from L_2 (nested guest) costs about 9.3 microseconds, which is about 33 times worse than L_1. And L_3 is the worst, with 232.7 microseconds, about 814 times worse than L_1.

The huge overhead noticed between L_1 and L_2, is the result of exit multiplication happening while the guest hypervisor handles the *vmcall* instruction. For this matter, we propose an optimization in the form of a patch [8] implemented in KVM ; it can be ported to other hypervisors as well. This patch optimizes the *vmcall* exit handling in L_0. The concept of this contribution is to avoid involving the nested hypervisor in the communication between the nested guest and the host, which removes any exit multiplication in the nested communication path using hypercalls. Mainly, this prevents the *vmcall* instruction, executed from L_2 (or L_n, with n>1), to be transmitted to L_1, since a single exit L_2 can cause many L_1 exits. This optimization processes *vmcalls* from nested layers with a single exit, for an additional 1-2% overhead compared to L_1, as also shown in table 4.1.

Table 4.1 *VMCALL* overhead from nested layers

	Baseline	Nested vmcall		Overhead	
	L_1	L_2	L_3	L_2	L_3
Default	286 ns	9.3 us	232.7 us	x33	x814
L_0 optimization	286 ns	289 ns	292 ns	1 %	2.1 %

4.9 Use Cases

This section shows how we used hypertracing to solve real world issues.

4.9.1 Early Boot Crashes

Kernel crashes are frequent during the development cycle ; kernel developers often use the serial console to inspect debugging messages like printed call stacks or registers values. A more sophisticated and modern way to debug kernel panics is tracing. Tracing enables developers to track back the prior events that led up to the crash. Ftrace implements a feature called ftrace_dump_on_oops. Enabling ftrace_dump_on_oops makes debugging crashes much easier by dumping the entire trace buffer to the console in ASCII format.

Kernel panics happening during the boot-up phase are much more difficult to investigate. The console output is the only available tool during boot-up, but it remains limited when non-trivial issues happen at a very early stage.
Ftrace is an internal Linux kernel tracer ; this interesting property makes this tracer most suitable to use for investigating kernel boot issues. We have submitted a kernel patch [7] to the Linux kernel to enable function tracing at a very early stage during boot-up. This contribution enables tracing and debugging during the early kernel boot-up. Moreover, we faced some challenges while developing this feature. We found no way to debug kernel crashes happening during the development of this feature. In early boot-up, the console output is not yet initialized, which makes it almost impossible to debug.

In a virtualized environment, the host system is fully available but does not have access to guest information, and the guest system has all the information but is in the initialization phase. Therefore, the goal is to use host collaboration to debug early guest initialization crashes by using paravirtualization. Hypercall is the most suitable paravirtualization technique at that point because it only uses registers and *vmcall* instructions to communicate. At an early stage, either memory sharing or networking cannot be used, so using

an infrastructure-less technique such as the hypercall is most appropriate for this particular configuration.

We used a combination of both static and dynamic code instrumentation techniques to insert our tracing hypercalls into the kernel code. For dynamic instrumentation, we used the mcount mechanism that can be enabled using a gcc compilation option called "-pg". We placed a hypercall inside the mcount callback function, which is invoked on every function entry. This way, we can get the call stack that led up to early initialization crashes. With static instrumentation, we instrumented some large functions to dump the values of some variables. We instrumented some inlined functions that were not handled by the mcount mechanism. We also instrumented some assembly code too, (hypercall was a perfect choice for that), because it only needs one instruction (*vmcall*) to perform tracing.

4.9.2 VM Boot-up

In any cloud platform, the procured VMs need some amount of time to be fully operational for cloud users. Cloud providers require time to select a host in their data centers on which to run the requested new VMs, for resources to be allocated (such as IP addresses) directly to the VM, as well as to copy or boot or even configure the entire OS image. Many surveys and blogs raised questions about these issues. More importantly, cloud users have also complained about this unexpected long provisioning time issue [30]. It hurts their cloud application performance and slows down their development productivity.
In fact, this dynamic process is inevitable to ensure cloud elasticity. A long undesirable latency during this process could result in a degradation of elasticity responsiveness, which will immediately hurt the application performance.

In a cloud system, three distinct stages are performed when a client requests to start a VM. In the first stage, the platform identifies a suitable physical node to allocate the requested VM. The second stage is based on transferring the VM image from the image store to the compute node. While in the final stage, the VM is booting on the physical node [45].
Most cloud providers have a centralized scheduler that orchestrates VMs provisioning. In this matter, the first stage can be completed within a few seconds. In the second stage, the network bandwidth and the image size do impact the transfer rate. Previous work [37] has performed a comparative performance study of VM booting on Rackspace, Azure and Amazon EC2. They concluded that the provider infrastructure greatly impacts the network transfer rate, which affects the overall launching process. Finally, the boot-up stage is mostly neglected in the cloud research. Researchers take for granted that this stage requires a small amount of resources.

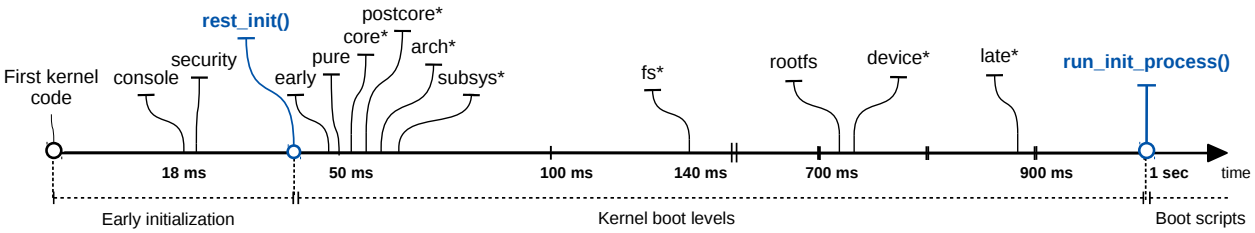

Figure 4.11 Linux boot-up sequences.

Figure 4.12 Boot-up stages.

The VM boot-up stage is performed like any normal Linux boot process, which is illustrated in Fig. 4.12. When the hypervisor executes the VMLAUNCH instruction, the BIOS starts by performing hardware checks. Then, it locates and loads into the memory the bootloader. Next, the boot loader loads the kernel binary into the main memory and perform a jump to the first kernel code located in *head.S*. Only processor CPU0 is running at this moment, and the kernel performs synchronous initialization with only one thread. Global data structures, CPU, virtual memory, scheduler, interrupt handlers (IRQs), timers, and the console are all initialized in a strict order. This part of the system is behaving like a real-time operating system, and runs fairly quickly. This behavior changes when the kernel runs the function rest_init() ; the kernel spawns a new thread to load built-in code (modules) and initialize other processors. The final phase happens when the kernel runs the init process (PID 1), the first userspace program. At this step, the system is preemptible and asynchronous.

Built-in modules are grouped by functionality into separate sections known as boot levels. These sections are defined as follows in a specific order : console, security, early, pure, core, post-core, arch, subsys, fs, rootfs, device and late. Fig. 4.11 shows the boot sequence order when the kernel boot-up happened in one second.
A built-in module can register itself in any boot level using the kernel initialization mechanism called *initcall* [25]. The boot order of each level is statically defined and identical across all Linux systems. However, the ordering of modules inside each level is determined by the link order. The blacklist feature allows users to prevent some built-in modules from loading during the boot-up, which enables runtime customization of the boot process.

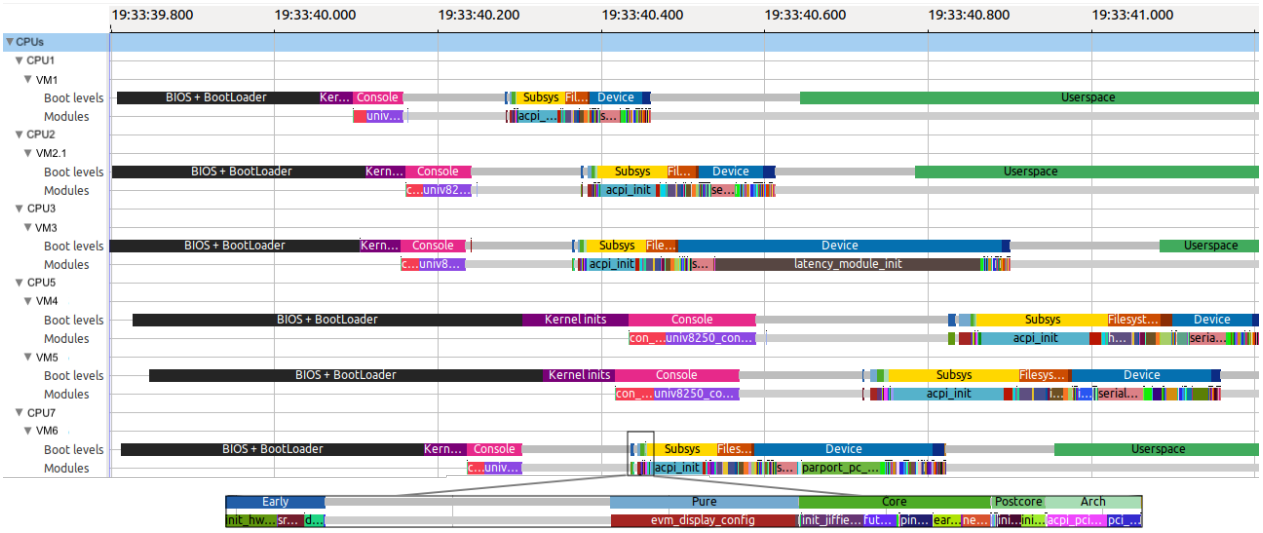

Figure 4.13 Tracing VMs boot-up levels and built-in modules.

Figure 4.14 Detecting kernel panics happening during a VM boot-up.

To be able to monitor issues happening during VM boot-ups, we have developed two Ftrace plugins [8]. The first plugin is the **bootlevel** tracer, which enables tracing boot level sequences. The second plugin is the **hypergraph** tracer. This technique provides the ability to trace guest kernel call stacks directly from the host. It works by offloading (through hypercalls) the guest sched_switch and function entry/exit events. This technique can also be used to debug other use cases like unexpected shutdown latencies, or kernel crashes. In this section, hypergraph would be configured to solely trace initcalls (built-in modules) entry & exit, which would help detect latencies during boot-up.

We produced two different experiments to test our method. In both experiments, we enabled host tracing using the following trace events : kvm_hypercall, kvm_exit, kvm_entry and kvm_write_tsc_offset.
The first experiment focuses on understanding the boot time in various scenarios where concurrent VMs are booted, whereas the second experiment addresses the case where a crash occurs while loading a specific module.

The result of the first experiment is shown in Fig. 4.13. In this experiment, we spawned at the

same time six VMs VM1, VM2.1 (nested VM within VM2), VM3, VM4, VM5 and VM6, each with one virtual CPU and 2 GB of memory, and pinned their vCPU0 to a dedicated physical processor except for VM4 and VM5, they both have been assigned to a single core (CPU5). The following VMs, VM1, VM2.1, VM3, VM4, and VM5 have identical kernel images. Their kernel was configured using the make target *localmodconfig*, which provides a way to disable modules that are not required by the system. In our system, this configuration produced a lightweight kernel image ; it contains 830 built-in modules loaded during the boot-up. VM1 and VM2.1 are booting on separate processors CPU1 and CPU2 respectively. But they still have different boot times, about 850 ms and 930 ms respectively. The VM2.1 is booting from nested level (L_2), this explain the overhead (about 9%) compared to VM1.

Next, we study a scenario where two concurrent VMs (VM4 and VM5) are booting on the same CPU. The boot time of VM4 and VM5 is 1.6 and 1.5 seconds respectively, about twice the time it took for VM1 to boot. Furthermore, in Fig. 4.13 we notice that in either VM4 or VM5, latency was present across all boot sequences. In fact, VM4 and VM5 were preempting each other all the time, and they shared the processor resources equitably through boot sequences.

In VM3, we implemented a custom module named latency_module_init loaded under the device boot level. This module introduces a 324 ms latency at the device level, which added a 35% overhead to the boot-up, and resulting in 1.3 seconds of boot time.

The boot time of VM6 was 1.14 seconds, VM6 was also configured using the make target *localmodconfig*. Additionally, we changed some configuration entries that were setup as "=m" by "=y." When a module is set with "=m," it will be loaded as an external module once the boot-up finished. By turning a configuration value from "=m" to "=y," the selected modules will be loaded during the boot-up at the device level. This change added 130 modules to be loaded by VM6 as compared to VM1, and resulted in 27% overhead for the boot-up.

The result of the second experiment is shown in Fig. 4.14. This experiment aims to verify the effectiveness of hypertracing toward kernel crashes. We implemented a module called crash_module_init ; we registered this module to load during device boot level. This module tries to access a non-existing memory which immediately causes a kernel panic. As a result, our monitoring technique detected this crash right away, as seen in Fig. 4.14.

4.9.3 VM Shutdown

We verified that the hypertracing method works for tracing late shutdown sequences while rebooting the VM. We used the hypergraph tracer to trace the kernel_restart(char *cmd)

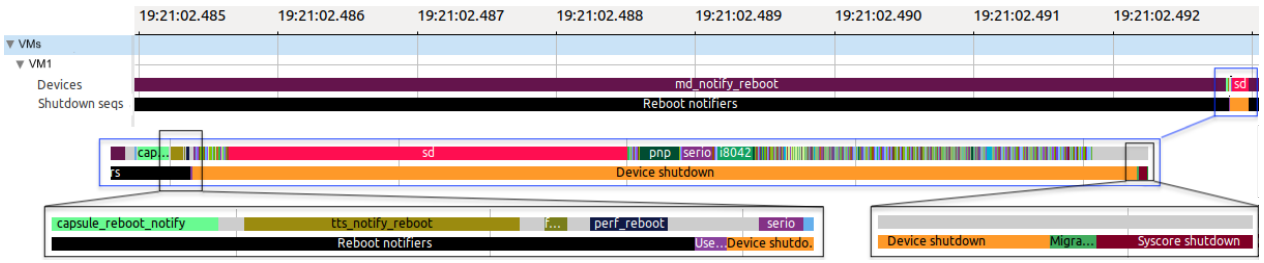

Figure 4.15 Tracing VM shutdown sequences.

function that is related to rebooting the system. The result is shown in Fig. 4.15.

The function kernel_restart is called after performing a reboot system call from userland. Then, the following sequence is performed :

Reboot notifiers : This step goes through the reboot notifiers list of hooks to be invoked for watchdog tasks. Fig. 4.15 shows that this step took about 1 second to complete. Most of the time was spent within the md_notify_reboot notifier call. While inspecting the Linux kernel source code, we found that the md_notify_reboot() function effectively performs a call to mdelay(1000) when at least one MD device (Multiple Device Driver or RAID) was used. In our case, the VM was using an MD device which led to the 1 second delays.

Disable usermod : This step ensures that no more userland code will start at this point.

Device shutdown : This step will release all devices on the system, about 340 devices were released in the experiment seen in Fig. 4.15. Among these devices, the sd device (driver for SCSI disk drives) took about 40% of the total time for this step.

Migrate to reboot cpu : In this step, the kernel forces the scheduler to switch all tasks to one particular CPU. Only a single CPU is running from this point on.

Syscore shutdown : This step performs some shutdown operations like disabling interrupts before powering off the hardware.

4.9.4 Rolling Upgrade

In a competitive market, cloud and online service providers guarantee the high availability of theirs platform services using Service Level Agreements (SLAs). Availability refers to the time that a service is operating correctly without encountering any downtime, which may be high but cannot reach 100%. For this reason, cloud providers commit to ensure an annual uptime percentage, like Amazon AWS which provides an uptime of at least 99.999%, equivalent to less than 6 minutes per year.

Based on recent studies [20, 15], software upgrades are the major cause of such downtime

(planned or unplanned) with up to 50% rates of upgrades failure.

Best practices recommend that industries avoid downtime using **rolling upgrades**. This mechanism enables upgrading and rebooting a single host or domain at a time through the data center [11]. However, this approach may result in temporary performance degradation caused by using the network when transferring the new release to the target nodes, and by having capacity loss while rebooting some nodes. Rolling upgrades are a good solution for resources with a high number of identical units, such as nodes in a cloud. There are other resources such as controller nodes or network switches where there are few units and the downtime associated with upgrading each unit is much more problematic, hence the importance in those cases of optimizing the upgrade shutdown and reboot cycle.

With the uprising of the computing as a service model, a big shift was noticed toward hardware simulation, enabling industries to test, simulate and validate their services using simulated hardware at low cost. Using a simulated environment improves development and ensures quality of the embedded software before deploying it to real hardware. A typical fault-tolerant embedded hardware unit is composed of one controller and two (or more) circuit boards defined as primary card and secondary card. The secondary card is used for resiliency purpose ; it takes control when the primary card is not available. VMs are used to simulate the hardware unit. A VM is needed to simulate the controller, and two others (SimVM1 and SimVM2) to simulate the primary and secondary cards.

A particularly interesting case we have seen in a product is the presence of some important performance degradation while performing a rolling upgrade within a simulator. An unexpected latency occurred while rebooting the SimVM in the context of an upgrade. Interestingly, this delay was not present when performing the same software upgrade on real hardware. As noted, the difference between the hardware and the simulation is the presence of virtualization and the application simulator.

Fig. 4.16 illustrates the sequence of the rolling upgrade. The first card to be upgraded is the secondary card SimVM2. The controller asks the primary card (SimVM1) to initiate the upgrade process when the target load (release) was delivered. Then, the steps 1, 2 and 3 are performed by the primary to prepare the secondary for the upgrade. Step 4 involves rebooting the secondary card, and it took about 40 seconds. When the secondary card is booted with the new load, SimVM1 sends a request to SimVM2 to switch its state to primary, then SimVM1 performs the reboot. After step 5, the primary card is now SimVM2, when SimVm1 is booted with the new load it becomes secondary. In the final step, SimVM2 sends to the controller a

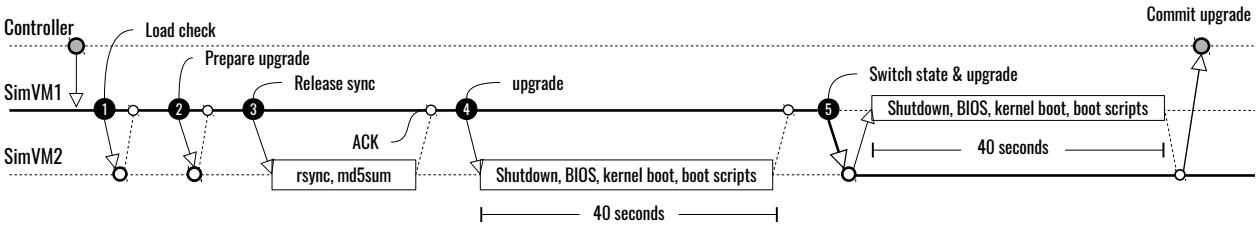

Figure 4.16 Performing rolling upgrade on the application simulator.

commit message which indicates that the upgrade was completed.

During the upgrade, the whole system becomes vulnerable and loses its redundancy (fault-tolerance) for about 80 seconds, which is a significant period of time. The boot-up and shutdown sequences are an important part of the upgrade and should be closely investigated.

We traced both the shutdown and boot-up using the hypertracing techniques previously presented (bootlevel and hypergraph tracers). As a result, we found no latency related to the kernel boot-up and the shutdown, they both took about 1 second each to complete. Normally, it is possible to enable monitoring tools after boot-up, but during the upgrade, the stored traces are destroyed, because of the unmounting operations that are performed to prepare for mounting the new release filesystem. To overcome this situation, we decided to use hypertracing to trace boot scripts initialization too. For this matter, we have developed another hypertracing tracer called **hypertrace** [8]. Since we did not know what caused this latency, we enabled some specific events that are presented as follows : scheduling events, system calls, timer and interrupts (IRQs) events.
Analyzing 40 seconds of a trace is very difficult, thus we decided to use the critical path analysis [26]. This analysis is already integrated into TraceCompass [1], and was developed by our research group [2]. The result of the analysis is shown in Fig. 4.17.

After the boot-up, the application simulator performs some logging during its initialization ; the first logged message was "Starting simulation application" which occurred 30 seconds later than the previous message. When analyzing the results shown in Fig. 4.17, the active path of the application simulator was blocked by the syslog-ng process for about 30 seconds, this explains the presence of a 30 seconds interval within the simulator logs.

The 30 seconds delay seems like a default configuration for a network timeout. This means that the syslog-ng is somehow offloading logs to an unavailable remote server. However, as we

1. http ://tracecompass.org
2. http ://dorsal.polymtl.ca

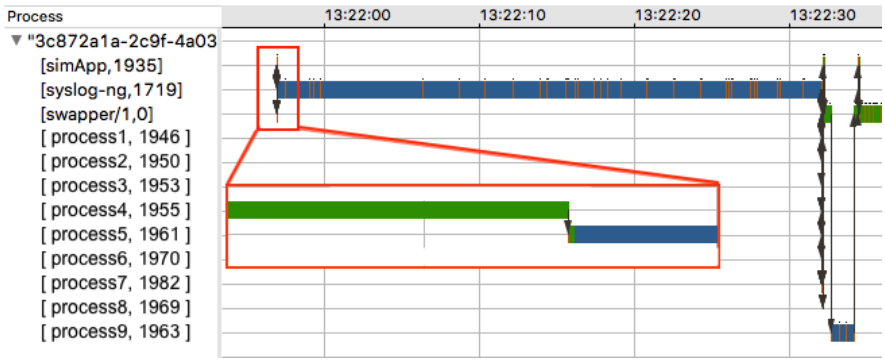

Figure 4.17 Active path of the application simulator. Some sensitive information like process's names were hidden, due to privacy concerns.

know, syslog would normally use the rsyslog process for remote logging, instead of syslog-ng, and there is no rsyslog process in the trace. While digging into the syslog configuration file, we discovered that syslog was configured for writing over an NFS mount, whereas the remote NFS server was unavailable. When syslog receives the timeout message 30 seconds later, it falls back to the default mode and then uses a local file for logging.

4.9.5 Virtualization Awareness

We verified that the hypertracing method works through a shared memory region. We used QEMU ivshmem (Nahanni) to enable a shared ring buffer between the guest and host. We enabled guest kernel tracing and configured it to write into the shared buffer. On each hypervisor trap, we record the hypervisor event (VMEXIT and VMENTRY) into the shared buffer. With this configuration, the guest could detect delays related to virtualization.

In this use case, we show how virtualization awareness could help us to find the root cause of latency from guest only. We executed sysbench as a CPU workload (prime numbers) inside the guest which had a single vCPU. Fig. 4.18 shows the results of guest tracing.

We have used three views to explain our findings. The control flow view shows that the sysbench process was running (Green color) on the CPU for the whole period of time. This is the kind of view that we usually get without enabling virtualization awareness. The virtual CPU view shows additional information, such as vCPU preemption states (Purple color) and VM transiting states (Red color) from VMX root mode or non-root. As expected, the vCPU was preempted many times during this experiment, this is a normal behavior. But we notice a long delay of 40 ms caused by a vm_exit with an exit reason of 1. The value 1 means that an external interrupt was triggered from the host to preempt the vCPU.
The virtual resources view shows which physical CPU was used by the guest. We can clearly

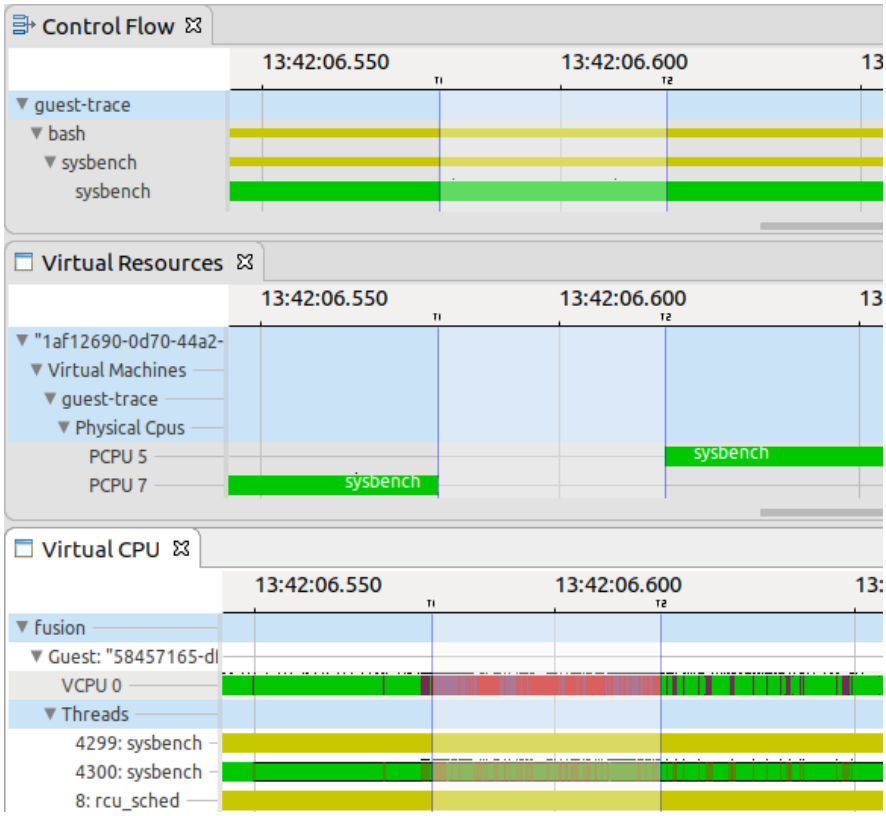

Figure 4.18 vCPU migration detection.

see that the guest vCPU0 was migrated from pCPU7 to pCPU5. Resulting into a 40 ms latency caused by the vCPU migration.

4.10 Overhead Analysis

In this section, we compare the overhead of our approach with the existing monitoring tools. Table 4.3 presents the added overhead of tracing boot-up sequences for existing tracing tools. Our bootlevel tracer achieved a very low overhead of 0.44% ; no existing tool can be compared to our tracer because none of them do trace boot levels. Bootgraph and function_graph tracer are both able to trace initcalls, similar to our hypergraph tool. Bootgraph uses the console to print the initcalls duration. Then it parses the dmesg output to retrieve the timings of the initcalls. The Ftrace function_graph tracer uses the mcount mechanism to trace function entries & exits ; each event will be recorded to the ring-buffer.
Hypergraph achieved the lowest overhead of about 0.52%, compared to both bootgraph and function_graph.

In order to get a clear understanding of the overhead involved when using hypertracing, we

Table 4.2 Comparison of multi-level tracing approach [24] with our hypertracing approach for synthetic loads

Benchmark	Baseline	Multi-level tracing	Hypertracing			Overhead (%)			
			Shared memory	Hypercall	Batching	Multi-level	Shared memory	Hypercall	Batching
File I/O (ms)	52.380	58.220	55.960	72.714	65.580	11.15	6.83	38.82	25.2
Memory (ms)	525.788	538.544	537.530	540.368	537.118	2.42	2.23	2.77	2.15
CPU (ms)	1380.34	1428.076	1421.426	1430.085	1426.404	3.45	2.97	3.6	3.33

Table 4.3 Comparison of existing boot-up tracing tools and our hypertracing approach

Boot-up tracing	Time (ms)	Overhead (%)
Baseline	734.43	-
Bootlevel	737.67	0.44
Hypergraph (initcalls)	738.28	0.52
Bootgraph (initcalls)	740.45	0.81
Ftrace function_graph (initcalls)	743.06	1.17

ran sysbench Disk I/O, CPU and Memory representative workloads to compare our approach with the multi-level [24] tracing approach. We enabled tracepoints that are related to scheduling and system call events inside the VM. We also enabled hypervisor events on the host such as kvm_exit kvm_entry and kvm_hypercall, which are used by all the approaches. Moreover, we enabled synchronization events that were needed for the multi-level approach to work. The results are presented in Table 4.2.

Hypertracing through shared memory has the minimal overhead among all approaches. Particularly in File I/O workloads, our approach incurs an overhead of only 6% because the shared buffer was configured to be consumed asynchronously from a different host CPU, unlike the multi-level approach which incurs an overhead of 11%, mostly related to I/O operations inside a VM. Hypertracing though hypercalls shows a significant overhead of 38%, due to the higher presence of sys_write events. Enabling event batching did significantly reduce the overhead from 38% to 25%, about a 70% improvement since we only used a minimal batching configuration to group system call entry and exit events. Using a more aggressive batching configuration will definitely reduce the overhead even more. With CPU and memory workloads, all the approaches had similar overheads, around 2% and 3% respectively. Hypertracing performed slightly better compared to multi-level when enabling event batching.

4.11 Conclusion

In this paper, we presented hypertracing, a paravirtualization technique that we used as new monitoring infrastructure for investigating virtual machines performance issues.

We presented the strong and weak points of existing inter-VM communication channel alternatives to TCP/IP. Then, we compared their throughput and concluded that memory sharing achieved higher efficiency. We then explained how we developed hypertracing using hypercalls, which enables debugging VMs within sensitives phases such as system crashes, boot-up and shutdown. We also showed how hypertracing can be used with the shared-memory-based approach for enabling virtualization awareness within VMs.

Finally, we proposed an L_0 optimization in the form of a patch to KVM. This contribution enables efficient nested paravirtualization, and allows our technique to monitor any nested environment without additional overhead. Current monitoring tools incur a significant overhead due to exit multiplication caused by I/O operations.

In the future, we aim to combine hypercall and memory sharing techniques to enable advanced hypertracing mechanisms. This configuration would combine the advantages of each channel, and thus enable complex issues to be solved.

CHAPITRE 5 DISCUSSION GÉNÉRALE

5.1 Retour sur le traçage du démarrage du noyau

Dans le chapitre 4, nous avons présenté l'utilisation du traceur hypergraph avec le filtrage activé pour capturer les **initcalls** lors du démarrage du noyau. Toutefois, d'autres fonctions peuvent être intéressantes à tracer lors du démarrage. Pour ce faire, il faudrait seulement mieux manipuler les configurations du filtrage pour éviter le traçage inutile des fonctions indésirables, conduisant à des surcoûts inattendus.

Dans la plupart des situations, les composantes responsables du démarrage sont méconnues des développeurs, et ces derniers se retrouvent dans l'obligation d'inspecter le code source du noyau pour extraire le nom des fonctions à tracer. L'inspection du code source est un long processus qui est pénible. De plus, la complexité du code noyau rend cette tâche encore plus ardue a réalisé. Pour améliorer la productivité des développeurs, nous proposons l'application de la technique **d'analyse dynamique** au démarrage du noyau.

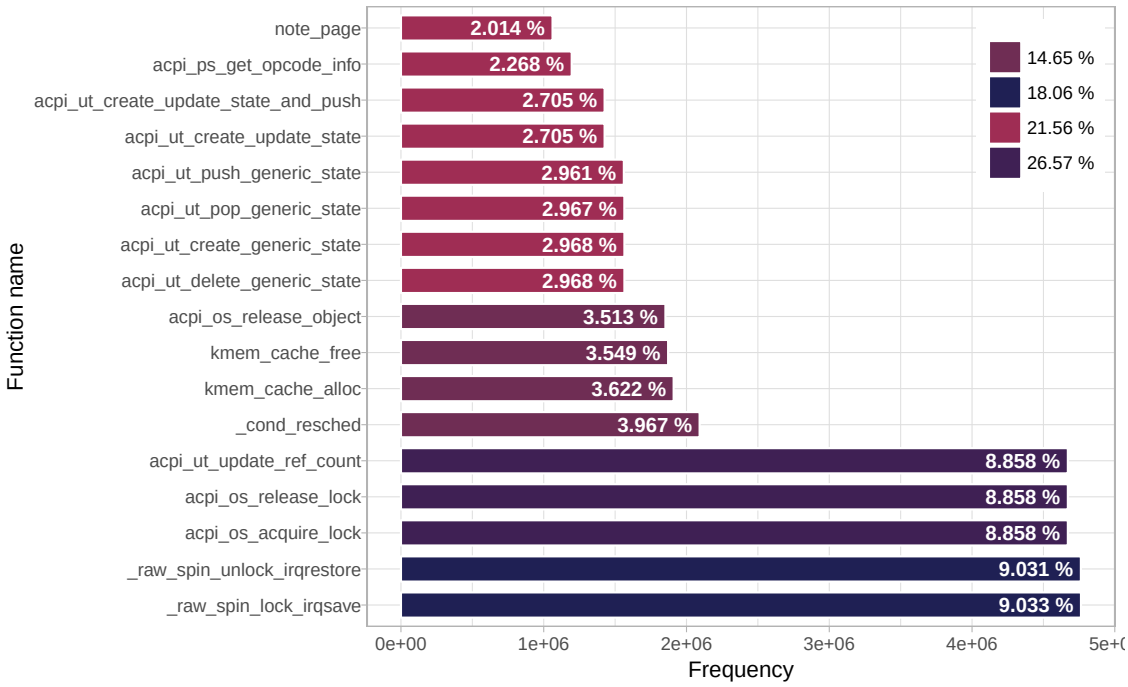

Figure 5.1 70% des appels les plus fréquents durant le démarrage

L'objectif de cette analyse et de permettre au développeur de comprendre l'exécution du programme à partir d'une seule exécution du démarrage. Pour ce faire, toutes les fonctions

doivent être tracées lors de cette analyse. La figure 5.1 nous montre les 17 fonctions les plus appelées durant le démarrage, représentant 70% des appels totaux. En analysant ces 17 fonctions, nous constatons la présence d'appels aux composants suivants : la gestion d'alimentation par l'API *ACPI*, l'utilisation de verrou "spin locks" et "mutex", et la gestion de la mémoire cache du noyau par *kmem*.

Nous avons produit quelques filtres pour contrôler le traçage de ces fonctions. La figure 5.2 présente le pourcentage des appels qu'on peut filtrer en utilisant les filtres proposés.

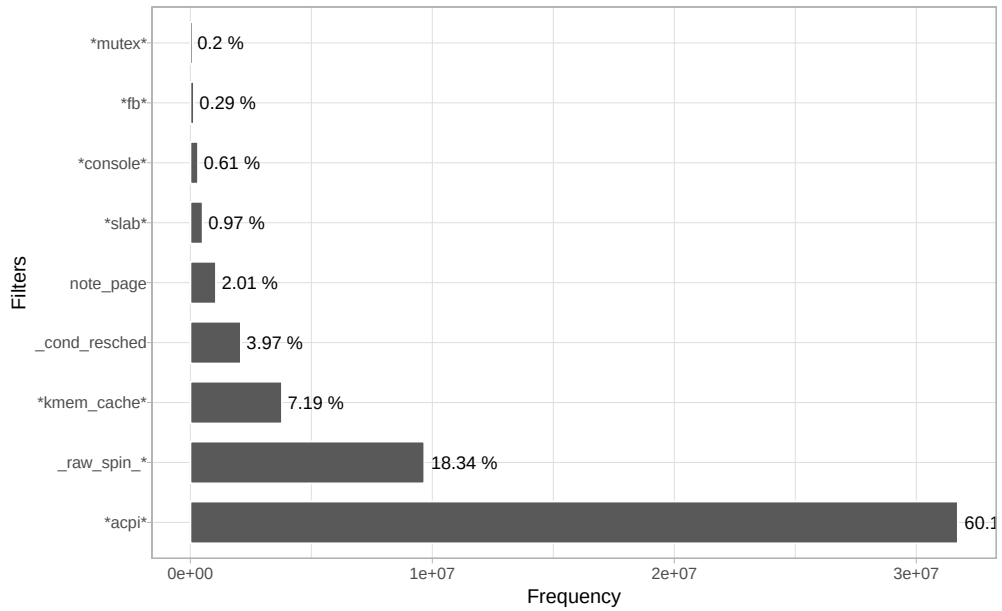

Figure 5.2 Pourcentage des appels filtrés avec l'utilisation des filtres

La figure 5.3 présente des résultats de performance concernant le traçage de fonctions du démarrage en appliquant une combinaison de techniques d'optimisation de filtrage et de groupement des évènements (*event batching*) présentées dans le chapitre 4.
Sans filtrage, le démarrage a duré 1min 20s, avec une trace d'une taille de 9.4 GB, tandis qu'avec le filtrage, la durée du démarrage est devenue 5.2 secondes. Pour améliorer encore plus la performance, nous avons appliqué l'optimisation de la technique pour combiner les évènements des fonctions d'entrée et sortie en 1 seul évènement (*event batching*), permettant ainsi de diminuer par un facteur de deux la durée du démarrage, qui devient 2.6 secondes.

Traçage dans hôte	Traçage dans le système invité			Événements	Taille de trace	Temps
	Traceur	Configurations				
-	**Baseline**	-		-	-	868.52 ms
	Function graph	Function entries & exits		168 M	Buffer size	22 secs
	Hypergraph	Function entries & exits		-	-	1min 08s
Lttng	Hypergraph	Function entries & exits		322 M	9.4 GB	1min 20s
		Event Batching		163 M	4.8 GB	40 secs
		Batching + Filtering		**3.2 M**	**207 MB**	**2.6 secs**
	Hypergraph & Bootlevel	Batching + Filtering & Boot level	early	20 K	1.4 MB	93 ms
			pure	5 K	0.3 MB	2.6 ms
			core	8 K	0.5 MB	4 ms
			postcore	5 K	0.4 MB	2.7 ms
			arch	32 K	2.1 MB	17 ms
			subsys	115 K	7.5 MB	143 ms
			fs	2.5 M	155 MB	1871 ms
			rootfs	10 K	0.7 MB	6 ms
			device	412 K	26 MB	410 ms
			late	258 K	16 MB	87 ms

Figure 5.3 Traçage du démarrage en appliquant les optimisations

CHAPITRE 6 CONCLUSION

Cette section présente une synthèse des travaux introduits dans ce mémoire. Nous allons ensuite énumérer les différentes limitations de la solution proposée. Finalement, nous conclurons ce mémoire avec les orientations futures.

6.1 Synthèse des travaux

Au cours de la réalisation de ce projet, nous avons développé plusieurs outils d'hypertraçage permettant d'analyser le comportement des systèmes invités soit dans la phase normale, soit dans les phases sensibles comme le démarrage et la fermeture du système. Nos outils d'hypertraçage sont développés et intégrés dans le traceur Ftrace. Nos investigations des problèmes de démarrage et fermeture du noyau Linux ont donnés naissance à trois contributions majeures. La première contribution permet le traçage des appels de fonctions très tôt lors du démarrage. La deuxième contribution est sous la forme d'un module Ftrace nommé *hypergraph*. Ce dernier permet de tracer (depuis l'hôte) tous les appels d'entrée et sortie des fonctions exécutées par le noyau invité, en utilisant le mécanisme *d'hypercall*. Cet outil nous a permis d'identifier avec précision quelle fonction du noyau invité est responsable des baisses de performance causées par la virtualisation. La troisième contribution est aussi un plug-in Ftrace nommé *bootlevel*. Cet outil permet de tracer les séquences de démarrage du noyau à faible coût.

Nous avons aussi proposé une méthode d'hypertraçage basée sur la mémoire partagée. Nous avons utilisé Nahanni QEMU pour activer un tampon circulaire partagé entre l'invité et l'hôte. Les deux systèmes invité et hôte écrivent simultanément dans le tampon partagé. Cette méthode nous a permis de rendre visible la virtualisation (*Virtualization Awareness*) chez le système invité, permettant ainsi de détecter les latences causées par la virtualisation depuis le système invité.

Nous avons aussi proposé une contribution à l'hyperviseur KVM. L'activation de l'hypertraçage depuis des machines virtuelles imbriquées cause des baisses de performance à cause du phénomène de la multiplication de sorties. Cette contribution permet d'annuler ce phénomène afin de mieux utiliser les outils d'hypertraçage depuis des niveaux de virtualisation imbriqués.

En résumé, ce projet a permis de rendre disponibles de nouveaux outils de traçage de machines virtuelles, et ainsi d'atteindre les objectifs définis dans le chapitre 1. Nous avons aussi

implémenté des analyses graphiques présentées dans le chapitre 4. Ces analyses nous ont permis d'investiguer les problèmes réels du démarrage et de la fermeture des systèmes invités utilisés en industrie.

6.2 Limitations de la solution proposée

Nous énumérons dans cette section les limitations des travaux proposés. L'utilisation de la technique de l'hypertraçage exige l'implication de plusieurs composantes, au niveau de la paravirtualisation et du traçage, afin qu'elle soit opérationnelle, ce qui rend cette technique difficile à adopter dans un environnement de production. L'utilisation *d'hypercall* dans l'hypertraçage permet d'activer des fonctionnalités d'analyse plus avancées, sauf que l'utilisation excessive des hypercalls est assez coûteuse. Les transitions entre VM et VMM sont considérées comme l'inconvénient majeur de cette technique. En effet, ces transitions peuvent polluer la cache du processeur (*CPU cache pollution*) réduisant grandement la performance du système.

L'hypertraçage se base sur des techniques de paravirtualisation qui nécessitent le chargement de nouveaux modules noyau dans le système d'exploitation invité. Le fait d'installer de nouveaux modules noyau peut décourager certains utilisateurs d'adopter l'hypertraçage pour des raisons de sécurité.

6.3 Améliorations futures

Il existe plusieurs opportunités d'amélioration des travaux proposés dans ce mémoire. En premier lieu, il y a une possibilité de récupérer le flot d'exécution du démarrage du noyau des systèmes invités sans utiliser la paravirtualisation. Au cours des investigations sur les problèmes du démarrage, nous avons découvert quelques détails importants à ce sujet. Premièrement, les séquences de démarrage du noyau se font de façon séquentielle. La même séquence se produit pour tous les types de démarrages et pour tous les types de distribution utilisant Linux comme système d'exploitation. Deuxièmement, le démarrage est exécuté par un seul fil d'exécution, sur le processeur principal CPU0. En prenant les deux points précédents en considération, et sachant qu'un système invité est représenté par des fils d'exécutions nommés vCPU (CPU virtuel) dans l'hôte, il est donc possible de tracer le flot d'exécution du vCPU0 au cours du démarrage directement depuis l'hôte. Ceci est possible en utilisant la technique d'instrumentation dynamique qui permettra d'instrumenter dynamiquement le code du démarrage puis de le désactiver une fois terminé.

Aussi, au lieu d'utiliser les techniques d'hypertraçage individuellement, il est possible de

combiner le hypercall et la technique de partage de mémoire dans un mécanisme d'hypertraçage avancé. Cette configuration permettra de bénéficier des avantages de chaque canal de communication, et ainsi de permettre une investigation plus poussée des systèmes invités.

RÉFÉRENCES

[1] Real time linux collaborative project. Linux Foundation Wiki, 2018.

[2] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM SIGOPS Operating Systems Review*, 40(5) :2–13, 2006.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

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

[5] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[6] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The turtles project : Design and implementation of nested virtualization. In *OSDI*, volume 10, pages 423–436, 2010.

[7] A. Benbachir. ftrace : support very early function tracing, January 2018.

[8] A. Benbachir. Online repository, January 2018.

[9] C. Biancheri and M. R. Dagenais. Fine-grained multilayer virtualized systems analysis. *Journal of Cloud Computing*, 5(1) :19, 2016.

[10] G. Brendan. Linux enhanced bpf (ebpf) tracing tools. *http ://www.brendangregg.com/ebpf.html. Consulté en Janvier*, 2018.

[11] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4) :46–55, 2001.

[12] J. Che, Q. He, Q. Gao, and D. Huang. Performance measuring and comparing of virtual machine monitors. In *Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on*, volume 2, pages 381–386. IEEE, 2008.

[13] K. H. Chung, M. S. Choi, and K. S. Ahn. A study on the packaging for fast boot-up time in the embedded linux. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 89–94. IEEE, 2007.

[14] T. Compass. Trace compass. *En ligne : http ://tracecompass. org*, 2018.

[15] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel. Staged deployment in mirage, an integrated software upgrade testing and distribution system. *ACM SIGOPS Operating Systems Review*, 41(6) :221–236, 2007.

[16] J. De Gelas and I. ESX. Hardware virtualization : the nuts and bolts. *AnandTech. Retrieved March*, 17 :2008, 2008.

[17] A. C. De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, 2010.

[18] M. Desnoyers. *Low-impact operating system tracing*. PhD thesis, École Polytechnique de Montréal, 2009.

[19] M. Desnoyers. Common trace format (ctf) specification (v1. 8.2). *Common Trace Format GIT repository*, 2012.

[20] T. Dumitraş and P. Narasimhan. Why do upgrades fail and what can we do about it ? : toward dependable, online upgrades in enterprise system. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, page 18. Springer-Verlag New York, Inc., 2009.

[21] J. Edge. Triggers for tracing. *http ://lwn.net/Articles/556186/. Consulté en Janvier*, 2018.

[22] F. C. Eigler and R. Hat. Problem solving with systemtap. In *Proc. of the Ottawa Linux Symposium*, pages 261–268, 2006.

[23] M. Gebai and M. R. Dagenais. Survey and analysis of kernel and userspace tracers on linux : Design, implementation, and overhead. *ACM Computing Surveys (CSUR)*, 51(2) :26, 2018.

[24] M. Gebai, F. Giraldeau, and M. R. Dagenais. Fine-grained preemption analysis for latency investigation across virtual machines. *Journal of Cloud Computing*, 3(1) :23, 2014.

[25] M. Gilber. Kernel initialization mechanisms, June 2005.

[26] F. Giraldeau and M. Dagenais. Wait analysis of distributed systems using kernel tracing. *IEEE Transactions on Parallel and Distributed Systems*, 27(8) :2450–2461, 2016.

[27] V. Goyal, N. Horman, K. Ohmichi, M. Soni, and A. Garg. Kdump : Smarter, easier, trustier. In *Linux Symposium*, page 167, 2007.

[28] P. Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B : System programming Guide, Part*, 2, 2011.

[29] D. Gupta, R. Gardner, and L. Cherkasova. Xenmon : Qos monitoring and performance profiling tool. *Hewlett-Packard Labs, Tech. Rep. HPL-2005-187*, pages 1–13, 2005.

[30] T. Hoff. Are long vm instance spin-up times in the cloud costing you money ?, January 2018.

[31] J. Hwang, S. Zeng, F. y Wu, and T. Wood. A component-based performance comparison of four hypervisors. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 269–276. IEEE, 2013.

[32] H. Jin, W. Cao, P. Yuan, and X. Xie. Xenrelay : An efficient data transmitting approach for tracing guest domain. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 258–265. IEEE, 2010.

[33] M. T. Jones. Virtio : An i/o virtualization framework for linux. *IBM White Paper*, 2010.

[34] E. Khen, N. J. Zaidenberg, A. Averbuch, and E. Fraimovitch. Lgdb 2.0 : Using lguest for kernel profiling, code coverage and simulation. In *Performance Evaluation of Computer and Telecommunication Systems (SPECTS), 2013 International Symposium on*, pages 78–85. IEEE, 2013.

[35] G. Lettieri, V. Maffione, and L. Rizzo. A study of i/o performance of virtual machines. *The Computer Journal*, pages 1–24, 2017.

[36] A. C. Macdonell et al. *Shared-memory optimizations for virtual machines*. University of Alberta, 2011.

[37] M. Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 423–430. IEEE, 2012.

[38] D. Marshall. Understanding full virtualization, paravirtualization, and hardware assist. *VMWare White Paper*, 2007.

[39] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu. Probing the guts of kprobes. In *Linux Symposium*, volume 6, 2006.

[40] S. McCanne and V. Jacobson. The bsd packet filter : A new architecture for user-level packet capture. In *USENIX winter*, volume 93, 1993.

[41] R. McDougall, J. Mauro, and B. Gregg. *Solaris performance and tools : DTrace and MDB techniques for Solaris 10 and OpenSolaris*. Prentice Hall„ 2006.

[42] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 13–23. ACM, 2005.

[43] H. Nemati and M. R. Dagenais. Virtual cpu state detection and execution flow analysis by host tracing. In *Big Data and Cloud Computing (BDCloud), So-*

cial Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)(BDCloud-SocialCom-SustainCom), 2016 IEEE International Conferences on, pages 7–14. IEEE, 2016.

[44] H. Nemati, S. D. Sharma, and M. R. Dagenais. Fine-grained nested virtual machine performance analysis through first level hypervisor tracing. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 84–89. IEEE Press, 2017.

[45] T. L. Nguyen and A. Lebre. Virtual machine boot time model. In *Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on*, pages 430–437. IEEE, 2017.

[46] S. Patni, J. George, P. Lahoti, and J. Abraham. A zero-copy fast channel for inter-guest and guest-host communication using virtio-serial. In *Next Generation Computing Technologies (NGCT), 2015 1st International Conference on*, pages 6–9. IEEE, 2015.

[47] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7) :412–421, 1974.

[48] K. T. Raghavendra. Kvm : Add documentation on hypercalls. Linux Documentation Source Code, 2012.

[49] K. K. Ram, J. R. Santos, and Y. Turner. Redesigning xen's memory sharing mechanism for safe and efficient i/o virtualization. In *Proceedings of the 2nd conference on I/O virtualization*, pages 1–1. USENIX Association, 2010.

[50] Y. Ren, L. Liu, Q. Zhang, Q. Wu, J. Wu, J. Kong, J. Guan, and H. Dai. Residency-aware virtual machine communication optimization : Design choices and techniques. In *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pages 823–830. IEEE, 2013.

[51] V. Romanovsky. Libvirt : Enable panic device notification, January 2018.

[52] S. Rostedt. Trace event documentation in the linux kernel. Noyau Linux version 4.15.

[53] S. Rostedt. Finding origins of latencies using ftrace. In *11th Real-Time Linux Workshop*, pages 28–30, 2009.

[54] S. Rostedt. trace-cmd : A front-end for ftrace. *LWN-Linux Weekly News-online*, 2010.

[55] S. Rostedt. Using the trace_event() macro. *http ://lwn.net/Articles/379903/. Consulté en Janvier*, 2018.

[56] R. Russell. virtio : towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5) :95–103, 2008.

[57] S. D. Sharma. An entertaining ebpf xdp adventure, 2017.

[58] S. D. Sharma, H. Nemati, G. Bastien, and M. Dagenais. Low overhead hardware-assisted virtual machine analysis and profiling. In *Globecom Workshops (GC Wkshps), 2016 IEEE*, pages 1–6. IEEE, 2016.

[59] D. Stolfa. Tracing virtual machines in real-time. Master's thesis, University of Rijeka. Faculty of Engineering., August 2017.

[60] C. V. Waldspurger. Introduction to virtual machines. 2010.

[61] Y. Yunomae. Integrated trace using virtio-trace for a virtualization environment. LinuxCon North America/CloudOpen North America,New Orleans, LA. Keynote presentation, 2013.