THÈSE DE DOCTORAT DE
L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunication et Électronique (Paris)

Présentée par

Clément DÉVIGNE

Pour obtenir le grade de :

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Exécution sécurisée de plusieurs machines virtuelles sur une plateforme Manycore

soutenue le 6 Juillet 2017
devant le jury composé de :

M. Alain GREINER	Directeur de thèse	UPMC/LIP6 (Paris)
M. Quentin MEUNIER	Encadrant de thèse	UPMC/LIP6 (Paris)
M. Franck WAJSBÜRT	Encadrant de thèse	UPMC/LIP6 (Paris)
Mme. Fabienne NOUVEL-UZEL	Rapporteur	INSA/IETR (Rennes)
M. Daniel CHILLET	Rapporteur	ENSSAT/IRISA (Rennes)
M. Pascal BENOIT	Examinateur	UDM2/LIRMM (Montpellier)
M. Bertrand GRANADO	Examinateur	UPMC/LIP6 (Paris)
M. Sébastien PILLEMENT	Examinateur	UDN/IETR (Nantes)

Remerciements

Je remercie tout particulièrement M.Alain Greiner, professeur des Universités à l'Université Pierre et Marie Curie (UPMC), pour avoir été mon directeur de thèse et pour m'avoir proposé un stage à la fin de ma première année de Master SESI. Ce stage a été le déclic qui m'a donné l'envie de poursuivre dans cette voie. Je remercie aussi M.Franck Wajsbürt, maître de Conférences à l'UPMC, pour m'avoir proposé ce sujet de thèse et suivi tout au long de ma progression.

J'adresse mes remerciements les plus sincères et j'exprime ma profonde reconnaissance à M.Quentin Meunier, maître de Conférences à l'UPMC, pour avoir été mon encadrant et ami pendant ces longues années de thèse. Sa disponibilité, son soutien sans faille, ses conseils éclairés m'ont été d'une aide très précieuse pour finaliser ce travail.

Je voudrais remercier les rapporteurs de cette thèse, Mme.Fabienne Nouvel-Uzel, maître de Conférences-HDR de l'Institut National des Sciences Appliquées de Rennes, et M.Daniel Chillet, Professeur des Universités de l'Université de Rennes, pour l'intérêt qu'ils ont porté à mon travail, pour leurs remarques constructives et pour leur participation au jury de ma thèse. Je remercie également M.Pascal Benoit, maître de Conférences-HDR de l'Université de Montpellier, et M.Bertrand Granado, professeur des Universités à l'UPMC, pour avoir accepté de participer à mon jury en tant qu'examinateurs. Mes remerciements s'adressent aussi à M.Sébastien Pillement, professeur des Universités de l'Université de Nantes, qui a accepté d'être président de mon jury.

Je tiens aussi à remercier M.Pirouz Bazargan, maître de Conférences à l'UPMC, et M.Philippe Coussy, professeur des Universités à l'Université de Bretagne-Sud, pour avoir accepté d'être membres de mon jury de mi-parcours et pour leurs conseils et remarques pertinentes en vue de l'écriture de ce manuscrit.

Je n'oublie pas les membres du département SoC du Laboratoire d'Informatique de Paris 6 (LIP6) pour l'environnement de travail agréable et chaleureux. Je remercie Julien Denoulet, Jean-Lou Desbarbieux, Karine Heydemann, Lionel Lacassagne, François Pêcheux, Farouk Vallette et tout particulièrement Emmanuelle Encrenaz pour son écoute et ses conseils. Je remercie aussi Mme.Marie-

Minerve Louërat, responsable du département SoC au LIP6, et Mme.Shahin Mahmoodian, gestionnaire de l'équipe AlSoC, pour leurs aides dans les démarches administratives.

J'exprime tous mes remerciements à mes collègues et amis du LIP6, Liliana Andrade, Cédric Ben Aoun, Inès Ben El Ouahma, Jean-Baptiste Bréjon, Alexandre Brière, César Fuguet, Marco Jankovic, Mohamed Karaoui, Laurent Lambert, Hao Liu, Joël Porquet, Vanessa Tran et Benoit Vernay, pour les discussions partagées, les idées échangées et tous les bons moments passés ensemble. Sans eux ces années auraient paru plus ternes.

J'adresse toute mon affection à ma famille. En particulier je remercie ma mère et mon père pour m'avoir fait comme je suis et pour m'avoir soutenu et encouragé au cours de toutes ces années. Je remercie mon frère et ma belle-sœur pour ces étés reposants sous le soleil toulousain, je n'oublierai jamais ces longues discussions sous le ciel étoilé de Villemur. Je remercie aussi ma tata Françoise ainsi que toute ma famille de Bourgogne.

Je remercie aussi l'Université pour sa richesse intellectuelle et pour son bien vivre avec les étudiants de diverses nationalités qui la compose.

Pour terminer j'adresse toute ma gratitude à toutes les personnes qui ont contribué à l'élaboration de ce travail et que je n'ai pas nommées.

Table des matières

Table des figures

Liste des tableaux

Chapitre

1 Introduction

Aujourd'hui le monde fait face à une explosion de la quantité de données numériques. Ces données proviennent entre autres des réseaux sociaux ou des nouveaux usages de l'informatique mobile comme les objets communicants. Les informations contenues dans ces données sont précieuses pour la société, qu'il s'agisse de données commerciales, économiques, environnementales ou concernant la santé et la vie privée des personnes. Il apparaît clairement que la problématique de la sécurité concernant l'accès aux informations, la protection des données privées et la protection de la vie privée sont des enjeux critiques. Ces données étant en constante augmentation, leur traitement nécessite de plus en plus de ressources de calcul.

Parallèlement, la puissance disponible pour le calcul ne cesse également d'augmenter, que ce soit en nombre de cœurs ou en puissance de calcul par cœur. En particulier, les architectures commerciales actuelles possèdent de plus en plus de cœurs, et tendent vers les architectures dites manycore, qui comprennent un grand nombre de cœurs de calcul intégrés sur la même puce. Les architectures manycore permettent d'exécuter de manière efficace les applications fortement parallèles, tout en fournissant la possibilité de paralléliser des applications différentes si les applications ne sont pas elles-mêmes suffisamment parallèles. Elles constituent en cela une ressource de calcul flexible, et il s'agit de la raison principale pour laquelle ces architectures sont en expansion. Une plateforme possédant un grand nombre de cœurs peut alors déployer de multiples applications et partager les ressources de calcul entre différentes piles logicielles indépendantes. Une granularité possible pour le partage des ressources se situe au niveau des systèmes d'exploitation : plusieurs systèmes partagent alors temporellement ou spatialement les cœurs de calcul. Cette technique, apparue dans les années 1970, est appelée virtualisation.

Le but de la virtualisation est de s'affranchir du matériel réel. Celui-ci est abstrait par un matériel virtuel offrant des caractéristiques fonctionnelles identiques, mais sur un ensemble de matériels différents. La virtualisation permet alors l'exécution d'un logiciel demandant un matériel spécifique sur un ordinateur possédant d'autres caractéristiques matérielles. Ce logiciel est alors encapsulé dans

une machine virtuelle, qui est gérée par un logiciel de confiance nommé hyperviseur. La virtualisation permet d'exécuter plusieurs machines virtuelles en parallèle, permettant ainsi de partager l'ensemble des ressources d'un ordinateur, par exemple la mémoire, les cœurs de calculs, ou les périphériques. Néanmoins, le partage de ressources entre plusieurs entités nécessite une certaine isolation entre ces piles logicielles indépendantes. Un avantage de la virtualisation est l'optimisation de l'utilisation des ressources physiques d'un ordinateur entre plusieurs utilisateurs indépendants.

1.1 Émergence du *Cloud Computing*

Récemment, le concept de *Cloud Computing* a vu le jour, et utilise massivement les techniques de virtualisation. Le *Cloud Computing* permet à un utilisateur d'accéder à des ressources distantes : ressources de stockage, de calculs ou même des applications. Il existe plusieurs types de services offerts par le *Cloud Computing*, dont les principaux sont :

- *Software-as-a-Service* (SaaS), où les utilisateurs peuvent utiliser des applications mises à disposition par un fournisseur, mais ne peuvent pas gérer l'infrastructure du système.
- *Platform-as-a-Service* (PaaS), où les utilisateurs peuvent déployer, sur la plateforme du fournisseur, leurs propres applications. Dans ce type de service, les utilisateurs ont seulement le contrôle sur leurs propres applications.
- *Infrastructure-as-a-Service* (IaaS), où les utilisateurs peuvent accéder à tout le matériel virtualisé. Dans ce type de service, les utilisateurs peuvent déployer des systèmes d'exploitation complets et ils doivent gérer l'ensemble de la machine virtuelle.

Ces différents types de services peuvent être déployés sur des *Cloud* différents. Il existe principalement trois types de *Cloud Computing* : le *Cloud* privé, mis en place pour une seule organisation ; le *Cloud* de communauté, où plusieurs organisations se partagent la plateforme ; et le *Cloud* public, qui est mis à disposition par un fournisseur de *Cloud* pour le large public.

Le choix entre ces différents types de *Cloud Computing* et de services (SaaS, PaaS ou IaaS) dépend des besoins de l'utilisateur en termes de flexibilité, de contrôle, de sécurité et de confiance dans le fournisseur du service. Par exemple, un utilisateur possède beaucoup plus de flexibilité et de contrôle sur un service de type IaaS que SaaS. De même, la sécurité offerte par un *Cloud* privé est bien plus importante que celle offerte par un *Cloud* public.

Comme le montre la figure 1.1, il est évident que le marché du *Cloud Computing* est en plein essor et qu'il risque de s'intensifier dans les prochaines années. Néanmoins, le partage de ressources, ainsi que le stockage de données en masse sur des plateformes pouvant être partagées et accédées par un grand nombre d'utilisateurs, posent un problème sur la confidentialité et l'intégrité des données circulant sur ces plateformes.

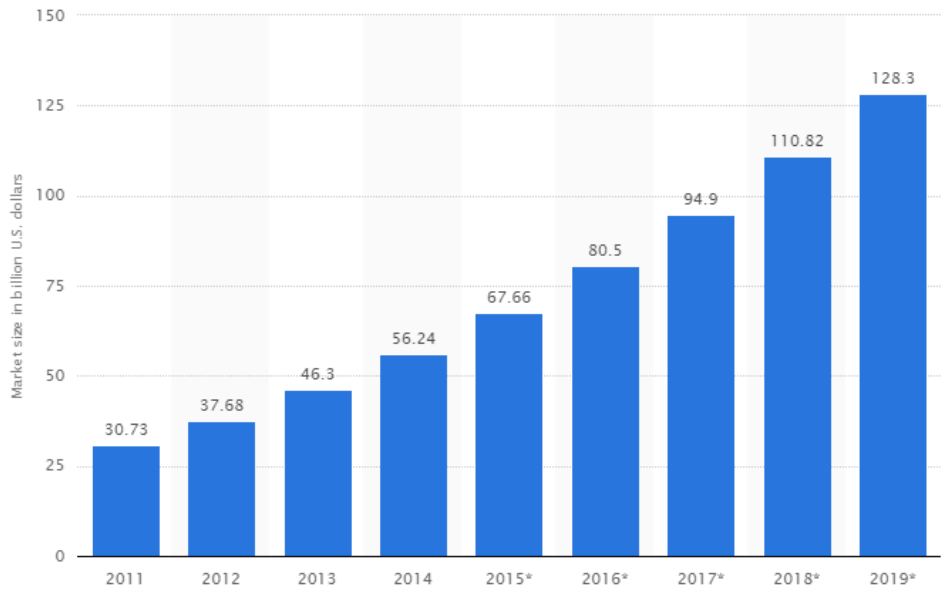

FIGURE 1.1 – Évolution du marché du *Cloud Computing* dans le monde de 2011 à 2019 (en milliard de dollars)
***Source** : Statista [1]*

1.2 Évolution des failles de sécurité logicielles

Bien que la virtualisation offre de nombreux avantages, ce nouveau concept introduit aussi un problème de sécurité lié au partage de ressources entre plusieurs utilisateurs et par le fait que le contrôle de la plateforme est effectué par un tiers logiciel : l'hyperviseur. Les hyperviseurs sont de plus en plus vulnérables aux attaques logicielles, et une attaque réussie sur l'hyperviseur peut être dévastatrice pour l'intégrité et la confidentialité des machines virtuelles s'exécutant sous son contrôle.

Ces attaques peuvent être catégorisées en fonction du niveau où elles exploitent les vulnérabilités :

- Au niveau des applications du système invité : ces attaques visent les applications utilisateur, par exemple en injectant du code malicieux pour dévier le flot d'instructions d'une application et faire exécuter du code de l'attaquant [2].
- Au niveau du noyau du système invité : ces attaques visent le système d'exploitation invité, par exemple des *rootkits* noyau qui permettent à un attaquant de prendre le contrôle du système [3].
- Au niveau de la couche de virtualisation : ces attaques utilisent les fonctionnalités offertes par la virtualisation pour attaquer d'autres machines virtuelles s'exécutant sur la plateforme [4].
- Au niveau de l'hyperviseur : ces attaques visent les vulnérabilités des hyperviseurs, permettant ainsi de gagner un contrôle complet sur l'ensemble de la plateforme et sur toutes les machines virtuelles s'exécutant dessus [5].

– Au plus bas niveau : ces attaques visent à attaquer la couche sous l'hyperviseur, comme le matériel, pour avoir un accès direct à la mémoire [6].

La figure 1.2 montre l'évolution du nombre de failles logicielles détectées au cours des dernières années. Comme nous pouvons le remarquer, les failles sont de plus en plus nombreuses, mettant ainsi en péril la sécurité des applications et des données. De nombreuses failles ont été découvertes dans les systèmes d'exploitation ou hyperviseurs alors que l'usage de plateforme de *Cloud Computing* était en plein essor. Ceci rend le problème de la confidentialité et de l'intégrité des données des utilisateurs bien réel. Par exemple, l'attaque réussie sur le *PlayStation Network* de la société Sony en avril 2011 à mené au vol de millions de données personnelles et bancaires.

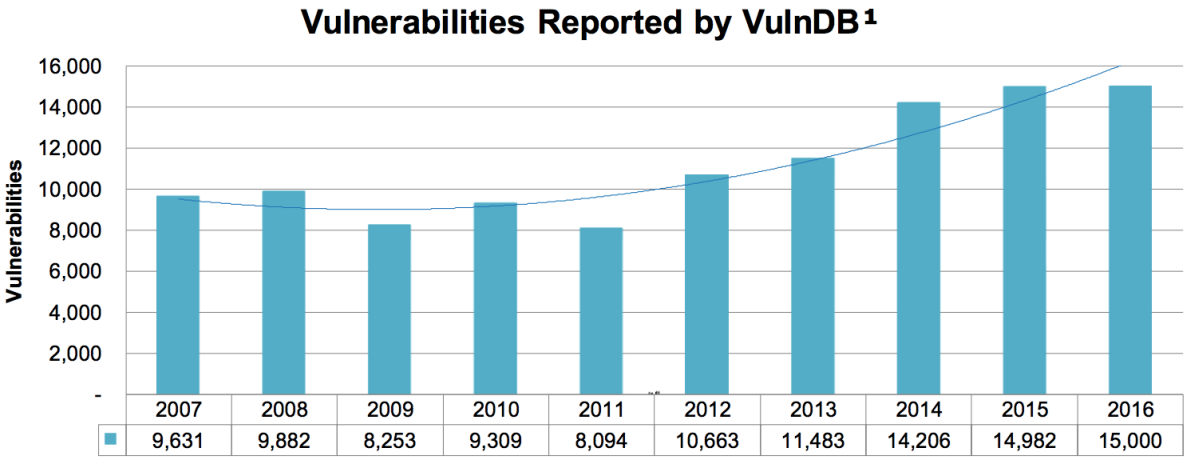

FIGURE 1.2 – Évolution du nombre de failles logicielles détectées entre 2007 et 2016
***Source** : Risk Based Security*

1.3 Problèmes et objectifs généraux de la thèse

Nous pensons que dans ce monde où les données traitées sont de plus en plus sensibles, il est impératif de garantir des propriétés de confinement et d'isolation entre les différentes machines virtuelles. Cette thèse tente de résoudre la problématique de sécurité liée à l'exécution de machines virtuelles sur des architectures manycore à mémoire partagée en définissant des mécanismes matériels et logiciels permettant d'offrir des garanties d'intégrité et de confidentialité pour les données traitées par les machines virtuelles.

Dans cette thèse, nous allons essayer de répondre aux trois questions ci-dessous :

1. *Comment faire cohabiter de façon sécurisée, en termes d'intégrité et de confidentialité, un large nombre de machines virtuelles s'exécutant sur un processeur manycore sans que les systèmes d'exploitation invités*

ne soient modifiés ?

2. *Comment empêcher l'hyperviseur d'être omniscient et omnipotent sur l'ensemble de l'architecture ?*
3. *Comment assurer un déploiement et une destruction dynamiques et sécurisés des machines virtuelles ?*

1.4 Organisation du document

Le chapitre 2 présente les enjeux de l'exécution sécurisée de machines virtuelles sur des architectures manycore et les problèmes que cette thèse cherche à résoudre. Nous présentons dans un premier temps l'évolution des architectures matérielles et l'apparition des architectures manycore. Dans un second temps, nous étudions le concept de virtualisation et son évolution au cours du temps. Dans un troisième temps, nous présentons notre modèle de menace. Enfin, nous énumérons les différents problèmes révélés par le contexte de l'étude.

Le chapitre 3 présente l'état de l'art relatif à la problématique. Dans un premier temps, nous présentons les différents types d'hyperviseurs ; dans un second nous présentons les mécanismes matériels existants permettant une virtualisation sécurisée ; enfin, nous étudions diverses architectures sécurisées existantes.

Le chapitre 4 présente nos différentes contributions. Nous présentons tout d'abord notre composant matériel en charge de l'isolation des machines virtuelles, puis nous présentons nos procédures de démarrage et d'arrêt des machines virtuelles. Enfin, nous expliquons dans la dernière section notre solution concernant le chiffrement du disque et l'intégration d'un crypto-processeur.

Le chapitre 5 présente les évaluations quantitatives et qualitatives de l'ensemble de nos contributions. Nous commençons par présenter les surcoûts matériels de l'ajout de nos composants puis les surcoûts en performances induit par nos contributions. Enfin, nous présentons dans la dernière section l'analyse de la sécurité de notre système.

Enfin, le chapitre 6 conclut en répondant aux questions posées dans la problématique et résume les contributions apportées. Nous présentons aussi les perspectives de ce travail.

Chapitre

2 *Problématique*

Contents

Ce chapitre présente le contexte de l'étude et les problèmes qui chercheront à être résolus par cette thèse. Dans un premier temps nous parlerons de l'évolution des architectures matérielles et plus particulièrement des architectures manycore. Nous étudierons plus spécifiquement une architecture manycore nommée TSAR pour *Tera Scale ARchitecture*. Dans un deuxième temps nous étudierons le concept de la virtualisation et les problèmes posés par cette technique. Nous définirons les différents types de virtualisation ainsi que leurs spécificités. Enfin nous définirons la problématique ainsi que les questions soulevées par le contexte de l'étude et dont cette thèse cherchera à répondre.

2.1 Évolution des architectures matérielles

Au cours des années les architectures matérielles ont connu une évolution. De simple architecture monocœur, nous sommes arrivés à des architectures pouvant intégrer jusqu'à plusieurs centaines de cœurs sur la même puce. Le nombre de cœurs intégrés sur une même puce n'est pas l'unique évolution, il y a eu aussi beaucoup d'améliorations concernant la distribution mémoire et l'optimisation du débit entre les cœurs et la mémoire principale, à travers l'utilisation des caches, permettant ainsi de repousser la limite du *memory wall*.

Cette section vise à présenter succinctement ces trois domaines d'évolutions concernant les architectures matérielles, à savoir l'évolution du nombre de cœurs, de la distribution mémoire et enfin des caches et de la cohérence des caches.

2.1.1 Nombre de cœurs intégrés

L'intégration *Very Large Scale Integration* (VLSI) continue de suivre la loi de Moore, ce qui a permis au cours des années l'intégration de plusieurs centaines de millions à plusieurs milliards de transistors sur un substrat de silicium. Cette diminution de la taille des transistors a permis entre autre de pouvoir créer des systèmes complets sur une même puce, appelés *System-on-Chip* (SoC). Les industriels tels qu'Intel se sont détournés des architectures monocœurs à cause de différents murs technologiques, par exemple le mur de la fréquence, en effet il devenait bien trop complexe de proposer des architectures monocœurs performantes. Les architectures dites multiprocesseurs sont alors apparues, utilisant des architectures moins complexes mais possédant plus de ressources de calculs permettant ainsi de tirer parti du parallélisme. Les prévisions stratégiques du rapport ITRS confirment cette tendance et montrent que le nombre de cœurs intégrés sur une même puce va encore augmenter au cours des prochaines années. Cette évolution en terme de nombre de cœurs a demandé une restructuration de l'informatique auparavant séquentielle vers l'informatique aujourd'hui parallèle. Ce parallélisme offre de nombreux défis en terme de scalabilité.

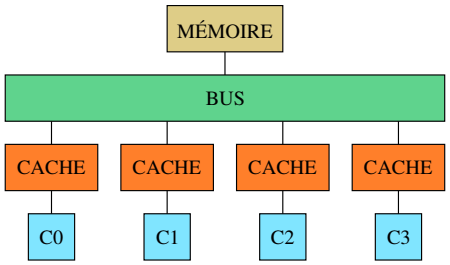

FIGURE 2.1 – Architecture SMP

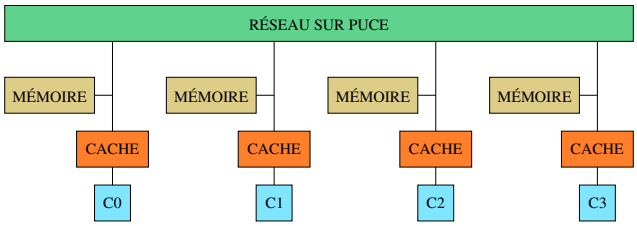

FIGURE 2.2 – Architecture NUMA

2.1.2 Distribution mémoire

La distribution de la mémoire a aussi connu une évolution. Au début, une mémoire unique était interconnectée à l'aide d'un bus aux différents cœurs. Ces architectures dites *Symmetric Multi Processor* (SMP) (2.1) ont présenté un problème de scalabitilité avec l'augmentation du nombre de cœurs. En effet, le fait de n'avoir qu'un unique banc de mémoire crée un goulot d'étranglement car il ne peut pas traiter les requêtes des différents cœurs à une vitesse suffisante. Pour pallier ce problème, le concept de mémoire logiquement partagée mais physiquement distribuée est alors apparu. Ce concept vise à distribuer la mémoire sur plusieurs nœuds. Cette méthode permet alors de décentraliser la mémoire pour avoir plusieurs bancs de mémoire qui se partagent l'espace adressable.

Ces architectures avec plusieurs bancs de mémoire ont temporairement résolu le problème, jusqu'à ce que le bus devienne lui-même un goulot d'étranglement. Les architectures dites *Non Uniform Access Memory* (NUMA) sont alors apparues. Ces architectures NUMA (2.2) se caractérisent par la présence de nœuds interconnectés à l'aide d'un réseau sur puce. Ces différents nœuds possèdent un nombre variable de cœurs et la mémoire partagée est physiquement distribuée dans chacun des nœuds. Ainsi, le temps d'accès à la mémoire varie en fonction de la distance à parcourir, autrement dit le nombre de nœuds à traverser.

2.1.3 Cache et cohérence

L'augmentation du nombre de cœurs accroît intrinsèquement le nombre d'instructions exécutées et donc la quantité de données traitées. En définitive nous augmentons le débit des échanges entre les cœurs et la mémoire. La présence d'une hiérarchie de cache entre les différents cœurs et la mémoire permet d'augmenter ce débit, en effet les caches permettent de stocker au plus près des cœurs des instructions ou des données fréquemment utilisées, permettant ainsi d'éviter des échanges trop fréquent entre la mémoire principale et les différents cœurs. La capacité des caches est bien évidemment très inférieure à la capacité de la mémoire principale mais occupent néanmoins une proportion élevée de la surface disponible d'une puce. Ils existent des caches de plusieurs niveaux possédant des caractéristiques particulières par exemple un cache de premier niveau est très rapide mais possède

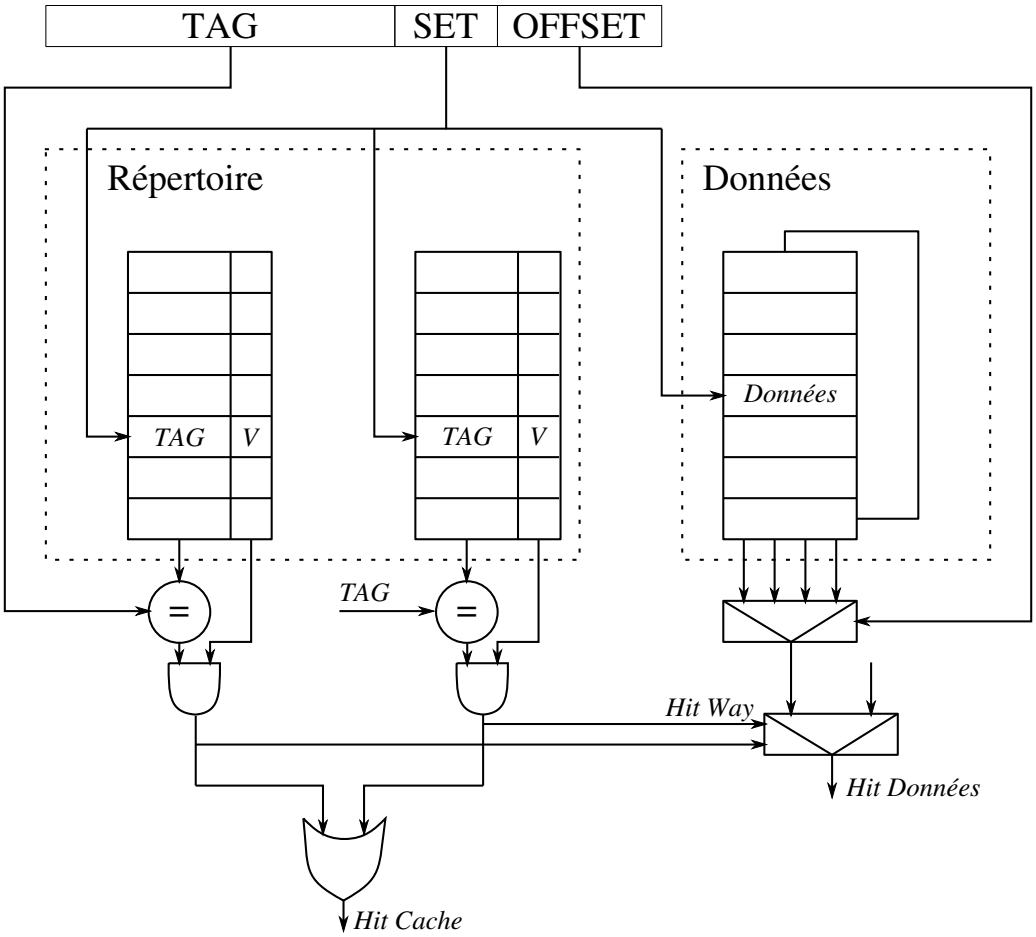

FIGURE 2.3 – Fonctionnement d'un cache

une capacité très faible alors qu'un cache de niveau supérieur va être beaucoup plus conséquent en terme de capacité mémoire mais est aussi beaucoup plus lent.

Les caches stockent des blocs de données contiguës en mémoire, ceux-ci sont de taille fixe et sont appelés lignes de cache. L'adresse d'un octet à l'intérieur d'une ligne de cache est nommée *offset*. L'adresse en mémoire de chaque ligne présente dans un cache est rangée dans un répertoire. Lorsqu'un cœur fait un accès mémoire le cache effectue une recherche, dans son répertoire, du numéro de ligne contenant la donnée permettant ainsi de savoir si le cache contient la donnée. Afin de réduire le temps d'accès, la position de la ligne dans le cache est définie par une partie de l'adresse de la ligne appelée *set*. Le reste de l'adresse de la ligne, nommé *tag*, est stocké dans le répertoire. Ce répertoire est indexé par *set* ou encore associatif par ensemble comme montré dans la figure 2.3.

Les caches se distinguent en deux principales familles en fonction de la stratégie concernant la propagation des écritures en mémoire. La première famille, nommée écriture simultanée (*write-through*), consiste à propager toutes les écritures vers la mémoire. La deuxième famille, nommée écriture différée (*write-back*) consiste à effectuer les écritures localement dans le cache et à les propager

lorsque cela est nécessaire.

Avec l'arrivée des architectures multiprocesseurs, l'utilisation des caches a posé des problèmes supplémentaires car des données pouvaient alors être partagées entre plusieurs cœurs. Ceci a alors induit une réplication des données au sein des différents caches et donc la possibilité d'avoir des versions différentes d'une même donnée en mémoire. La cohérence des données devient alors un problème important.

La cohérence des données dans les architectures multiprocesseurs peuvent être de deux types : une cohérence gérée en logicielle ou alors en matérielle. En ce qui concerne la cohérence maintenue en logicielle, le programmeur est responsable de la cohérence des données et doit donc invalider par lui-même les lignes de caches partagées entre plusieurs cœurs lorsque cela est nécessaire, par exemple à la suite d'une écriture par un cœur. La cohérence gérée par le matériel est complètement transparente pour le programmeur car c'est le système matériel qui va alors invalider les lignes de caches. La cohérence garantie par le matériel a cependant un prix en terme de surface et de complexité des contrôleurs de cache mais présente de très bonne performance et permet de simplifier le travail des programmeurs, c'est pourquoi cette technique est généralement utilisée.

2.2 Architectures manycore

Les architectures manycore sont des architectures intégrant des centaines de cœurs sur une même puce. De telles architectures utilisent des cœurs peu complexes permettant ainsi de maximiser la performance par Watt : selon la loi de Pollack, un doublement de la logique dans le processeur apporte un gain supplémentaire en performance qui n'est que de l'ordre de 40% [7]. En 2008, D. Woo *et al.* ont également montré à l'aide de modèles analytiques, que pour atteindre le meilleur rendement énergétique à l'intérieur d'un processeur, il fallait adopter une approche utilisant beaucoup de petits cœurs [8]. Ces architectures manycore sont généralement *clusterisées*, les clusters étant interconnectés entre eux *via* un réseau sur puce [9, 10]. Chaque cluster (figure 2.4) contient généralement un ou plusieurs cœurs et quelques périphériques, le tout connecté par un interconnect local. Mis à part le très bon ratio entre la performance et la puissance consommée, l'avantage principal des architectures manycore est leur redondance. Cette redondance permet la désactivation des cœurs inactifs ainsi que la tolérance aux pannes – qu'elles fassent suite à la fabrication ou à l'usage du circuit – par la désactivation des cœurs fautifs tout en gardant actifs les cœurs non défaillants.

Les architectures manycore diffèrent sur la façon dont les cœurs communiquent, à la fois à l'intérieur d'un même cluster ou avec un cluster distant. Certaines architectures utilisent des interfaces spécialisées (e.g [11]) ou bien des tampons matériels dédiés pour faire communiquer deux cœurs alors que d'autres supportent la mémoire partagée. Parmi les architectures utilisant la mémoire partagée, certaines possèdent des mécanismes matériels pour assurer la cohérence mémoire [12, 13] et

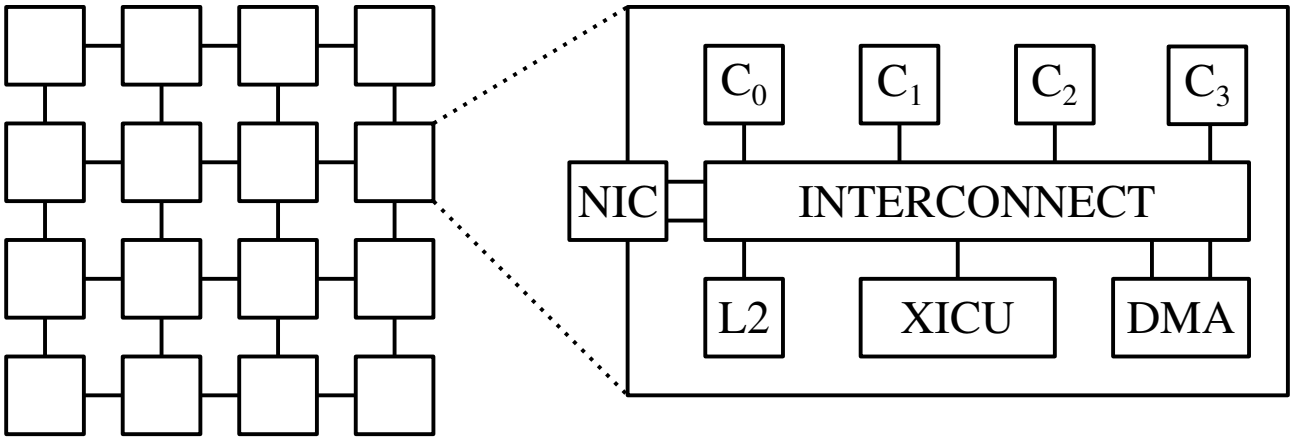

FIGURE 2.4 – Architecture Manycore

d'autres non [14].

Nous croyons que les architectures manycore doivent offrir une mémoire partagée cohérente par le matériel pour pouvoir supporter l'exécution de systèmes d'exploitation généralistes (par exemple Linux ou NetBSD). Dans un contexte de *cloud computing*, il est vrai qu'il n'est pas nécessaire d'allouer toutes les ressources à un même utilisateur, mais il faut néanmoins offrir un minimum de cœurs pour permettre d'avoir une puissance de calcul convenable puisque les cœurs sont moins performants.

En raison de leur structure, les architectures manycore sont capables d'exécuter plusieurs applications en parallèle et donc de permettre le traitement d'un large flux de données. Cependant, il faut pouvoir garantir des propriétés de sécurité à ces applications en particulier des garanties d'intégrité, de confidentialité et de disponibilité. De tels besoins doivent être conçus comme une part entière du système de traitement de l'information et ne doivent pas être considérés après conception par le simple ajout de matériel ou de logiciel cryptographique. Autrement dit, la sécurité des données doit être considérée comme une contrainte de conception et les architectures de systèmes doivent la garantir intrinsèquement.

En effet nous pensons qu'il est important de ne plus voir la sécurité comme une extension de l'architecture car cela se traduit souvent par un rajout de couches. Les interactions entre l'architecture de base et les rajouts de sécurité peuvent alors rentrer en collision et apportées des bogues au sein du système mettant alors en péril la sécurité du système complet. De plus il est à notre sens plus simple de garantir la sécurité d'un système si celui-ci a été conçu pour.

Il est donc indispensable de penser les architectures manycore en intégrant dès aujourd'hui la dimension sécuritaire.

2.3 L'architecture TSAR

L'architecture *Tera Scale ARchitecture* (TSAR) [15] est celle d'un processeur à mémoire partagée, à cohérence de cache garantie par le matériel, conçue au sein du Laboratoire d'Informatique de Paris 6 (LIP6). Elle est issue d'un projet Européen dirigé par la société BULL. Cette architecture permet d'exécuter des systèmes d'exploitation standards comme Linux ou NetBSD, et a été développée conjointement avec un système d'exploitation spécifiquement adapté aux manycores, nommé ALMOS [16]. Cette architecture est conçue pour passer à l'échelle jusqu'à 1024 cœurs, mais n'adresse nullement la problématique de la sécurité. Dans cette thèse, nous nous servirons de TSAR comme base, que nous chercherons à enrichir dans cette optique de sécurité.

2.3.1 Composition des clusters

TSAR est une architecture clustérisée utilisant un *Network-On-Chip* (NOC) et possédant une topologie en *mesh* 2D. Chaque cluster possède des coordonnées (x, y) déterminées par les 8 bits de poids fort de l'adresse physique.

Comme le montre la figure 2.5, présentant l'architecture TSAR, il existe deux types de clusters, le cluster dit *Input/Output* (I/O) contenant les périphériques externes et les clusters standards. Les clusters standards sont répliqués alors que le cluster I/O est unique.

Ce dernier contient entre autres des périphériques externes tels qu'un contrôleur de disque (IOC) ou alors un contrôleur de terminal (TTY). Il contient aussi un composant *Input Output Programmable Interrupt Controller* (IOPIC), permettant de transformer des interruptions matérielles en interruptions logicielles, car les fils d'interruption des périphériques externes sont connectés à l'IOPIC du cluster I/O. L'IOPIC est en charge d'envoyer des messages via le NOC en direction des concentrateurs d'interruption (XICU) contenus dans les clusters standards. Dans le cluster I/O on peut trouver aussi une *Read Only Memory* (ROM) contenant le code de *boot* de la machine. Un réseau I/O permet d'interconnecter tous les composants avec un accès vers le réseau des RAMs (réseau XRAM) et le réseau global (réseau L1/L2).

Les clusters standards contiennent les éléments suivants :

- Quatre cœurs MIPS-32 avec leur *Memory Management Unit* (MMU) et leur premier niveau de cache (L1), séparé entre instructions et données. La cohérence du cache L1 est entièrement gérée en matériel. Les *miss* dans le *Translation Lookaside Buffer* (TLB) sont aussi gérés en matériel.
- Un second niveau de cache (L2), qui est en charge d'un segment de l'espace mémoire de la machine. En particulier le cache L2 est responsable de la cohérence des copies dans les différents caches L1, pour les lignes contenues dans ce segment.

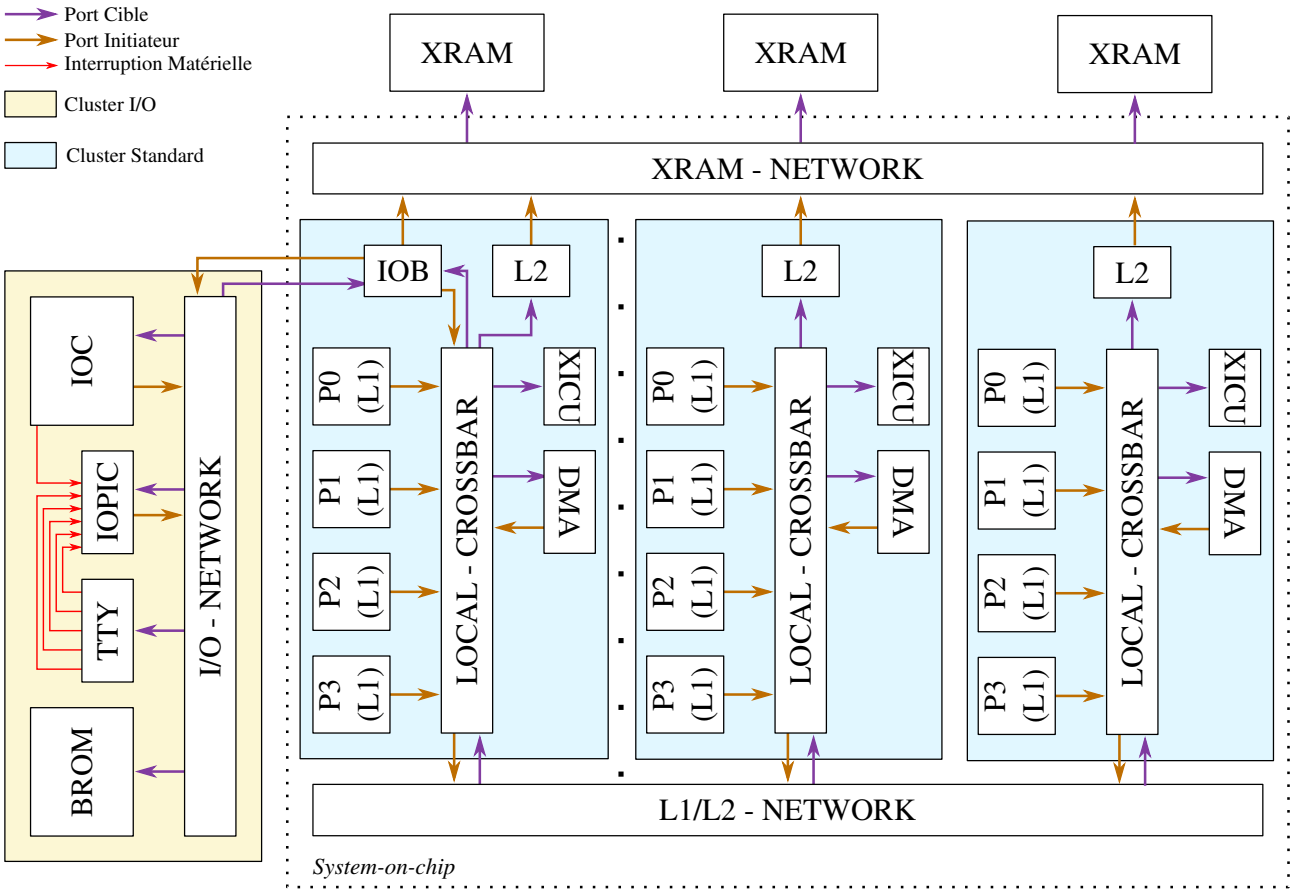

FIGURE 2.5 – Architecture TSAR

- Un contrôleur *Direct Memory Access* (DMA).
- Un contrôleur d'interruptions incluant des fonctions de *timer* (XICU). Ce composant gère deux types d'interruptions :
  - les interruptions matérielles (HWI) provenant des composants internes au cluster, typiquement le DMA.
  - les interruptions logicielles (WTI) provenant de l'IOPIC, celles-ci étant activées après des écritures, par l'IOPIC, dans des registres spéciaux de l'XICU. Ce type d'interruption est aussi utilisé pour implémenter le mécanisme d'*Interruption Inter-Processeur* (IPI).
- Un *crossbar* local permettant d'interconnecter ces composants avec un accès vers le réseau global (réseau L1/L2) par le biais d'un routeur.
- Un composant *Input/Output Bridge* (IOB), se trouvant dans le cluster de coordonnées (0, 0), qui permet l'accès au cluster contenant les périphériques.

2.3.2 Distribution de l'espace mémoire

TSAR gère un espace mémoire de 1 To, les adresses physiques sont donc sur 40 bits. Les 8 bits de poids fort sont réservés pour déterminer les coordonnées du cluster et les 32 bits de poids faible désignent le décalage (*offset*) dans le cluster. L'espace mémoire est segmenté, chaque segment est associé à un cluster et plus particulièrement chaque cache L2 de chaque cluster est en charge de ce segment. Dans la configuration maximale, nous avons donc 256 clusters et chacun des clusters gère un segment de 4 Go de mémoire physique. La figure 2.6 présente la distribution de l'espace mémoire pour une architecture TSAR à 4 clusters.

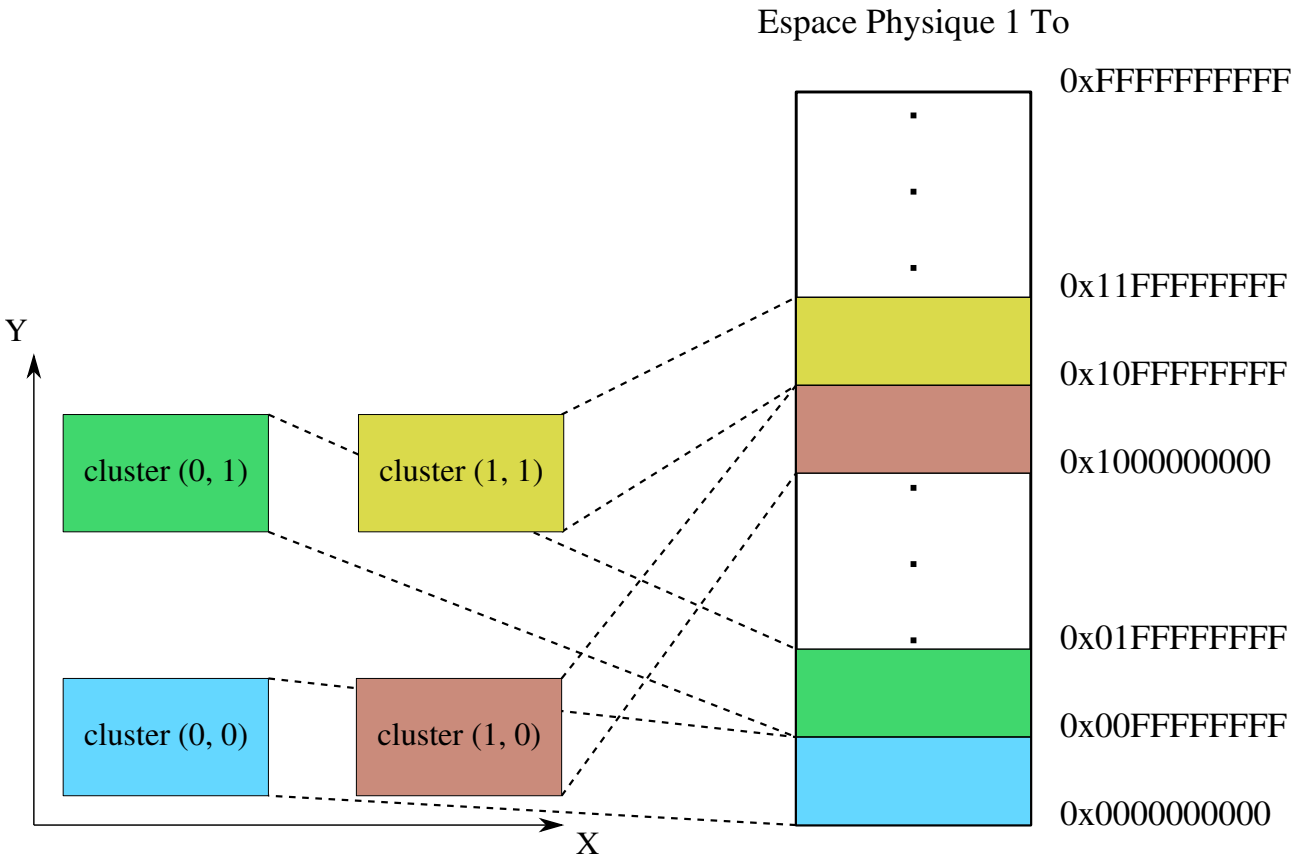

FIGURE 2.6 – Distribution de l'espace mémoire pour une architecture TSAR à 4 clusters

2.3.3 Protocole de cohérence de l'architecture TSAR

Le protocole de cohérence de cache mis en place dans l'architecture TSAR, nommé DHCCP pour *Distributed Hybrid Cache Coherence Protocol*, est un protocole basé sur le principe d'un répertoire global distribué. Chaque ligne de cache présente dans un cache L2 peut avoir n copies dans les caches de premier niveau. Pour chaque ligne, le répertoire du cache de second niveau contient soit l'identifiant des caches L1 possédant les copies au travers d'une liste chainée, soit le nombre de copies si le nombre

de copies n dépasse un certain seuil.

Le protocole DHCCP implémente une stratégie d'écriture immédiate *write-through*, autrement dit toutes les écritures effectuées par un cœur sont immédiatement envoyées au cache de second niveau responsable du segment adressé par l'écriture. Ainsi une ligne de cache est toujours à jour au niveau du cache L2.

Il existe au sein du protocole DHCCP deux directions pour les requêtes de cohérence : soit celles-ci sont envoyées par un cache L1 en direction d'un cache L2, soit l'inverse. Nous définissons cinq types de transaction de cohérence dans DHCCP. Celles-ci sont présentées dans la figure 2.7.

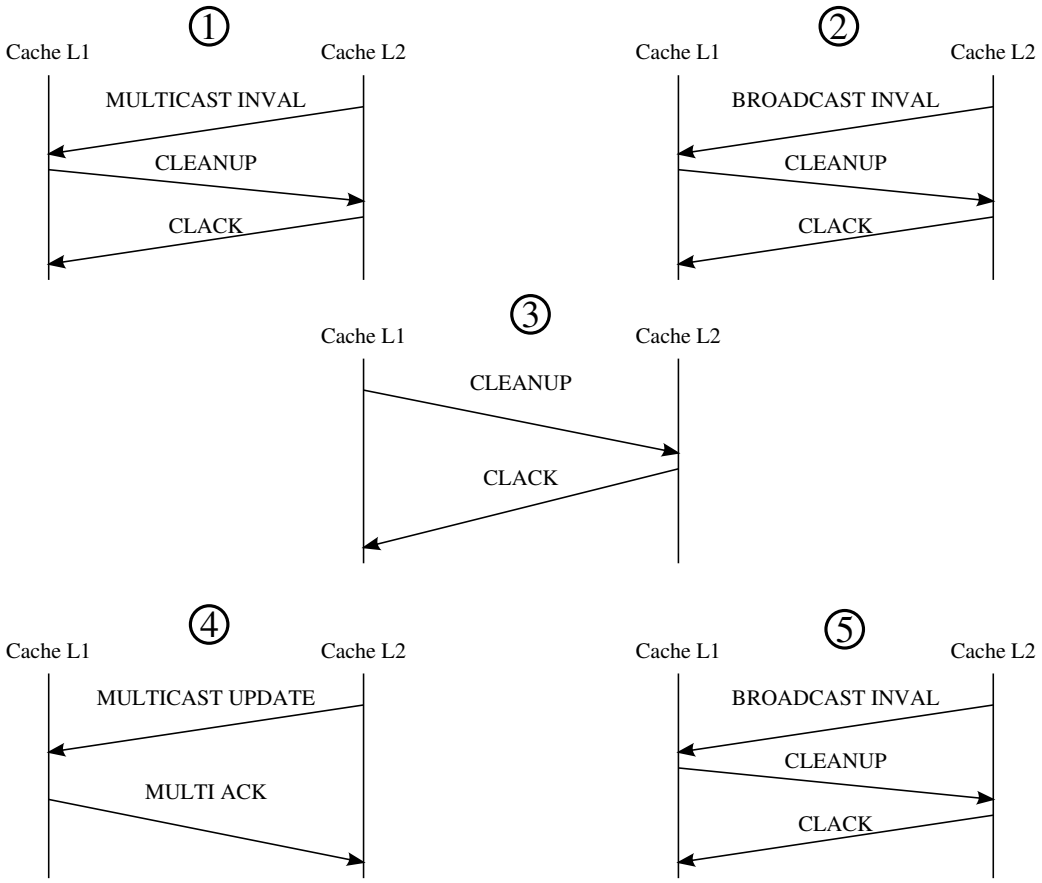

FIGURE 2.7 – Les différentes transactions de cohérence au sein de DHCCP

Transaction 1 : une fois que toutes les lignes d'un même ensemble (*set*) sont occupées, le cache L2 a besoin de libérer une case pour pouvoir rajouter une nouvelle ligne. Pour cela, le cache L2 décide d'évincer une ligne. Si celle-ci possède des copies au sein de différents caches L1 et que le nombre de copies est inférieur au seuil – autrement dit que la ligne est en mode *liste chainée* – alors le cache L2 doit envoyer une requête d'invalidation aux différents caches L1 possédant une copie. Cette requête d'invalidation, nommée *multicast invalidate*, est principalement composée de l'adresse de la ligne à invalider. Lorsque les caches L1 reçoivent cette invalidation, ceux-ci vont à leur tour évincer la ligne

de leur cache et envoyer un message *cleanup* au cache L2 responsable de la ligne. La requête *cleanup* contient principalement l'adresse de la ligne évincée par un cache L1. À la réception de ce *cleanup* le cache L2 va savoir qu'il correspond à une réponse à l'invalidation postée plus tôt et décrémenter un compteur d'attente de réponse. Pour finir le cache L2 va répondre au *cleanup* du cache L1 par un *clack*, pour *cleanup acknowledgement*.

Transaction 2 : cette transaction est lancée dans la même situation que celle de la transaction 1, mais dans le cas où le nombre de copies est supérieur au seuil. Ainsi la ligne ne se trouve plus en mode *liste chainée*. Le cache L2 va alors envoyer un *broadcast invalidate* à tous les caches L1 de la plateforme. Les caches L1 vérifient à la réception de ce *broadcast* s'ils possèdent la ligne. Si c'est le cas, ils vont simplement évincer la ligne et envoyer un *cleanup*. Le cache L2 compte alors le nombre de *cleanup* reçus pour cette ligne afin de connaitre la fin de la transaction *broadcast invalidate*. Chaque cache L1 reçoit un *clack* en réponse à son *cleanup*.

Transaction 3 : cette transaction est utilisée lorsqu'un cache L1 décide d'évincer une de ses lignes. Cette transaction permet de signaler au cache L2 responsable de la ligne que ce cache L1 ne possède plus de copie de la ligne. Il s'agit du même message que le *cleanup* qui sert de réponse à une transaction d'invalidation *multicast invalidate* ou *broadcast invalidate*.

Transaction 4 : cette transaction a lieu lorsqu'un cache L2 reçoit une écriture sur une ligne X partagée par plusieurs caches L1 en mode *liste chainée*. Le cache L2 va alors envoyer un *multicast update* vers les caches L1 possédant une copie de cette ligne. Cette requête *multicast update* contient principalement l'adresse de la ligne à mettre à jour ainsi que les nouvelles valeurs de la ligne. À la réception du *multicast update* les caches L1 mettent à jour la ligne avec les valeurs contenues dans la requête et envoient une réponse *multi ack* au cache L2 responsable de la ligne mis à jour.

Transaction 5 : cette transaction a lieu lorsqu'un cache L2 reçoit une écriture sur une ligne X partagée par plusieurs caches L1 en mode *compteur*. Le cache L2 va alors envoyer un *broadcast invalidate* vers tous les caches L1. Cette requête *broadcast invalidate* contient principalement l'adresse de la ligne à invalider. À la réception du *broadcast invalidate* les caches L1 vérifient qu'ils ont la ligne dans leur cache puis l'invalide et envoient un message *cleanup* au cache L2 responsable de la ligne. Le cache L2 compte le nombre de *cleanup* qu'il reçoit sur cette ligne, lorsque qu'il reçoit tous les *cleanup* la transaction est terminée. Pour chaque *cleanup* reçu le cache L2 envoie un *clack* vers le cache L1 responsable du *cleanup*

2.4 Virtualisation

Le concept de virtualisation est omniprésent en informatique. Le but de la virtualisation est de s'affranchir du matériel réel, qui est alors remplacé par un matériel virtuel offrant des caractéristiques

fonctionnelles identiques sur un ensemble de matériels différents. La virtualisation a été inventée au début des années 70 et visait au départ le marché des serveurs [17]. Un serveur virtualisé pouvait en effet offrir plusieurs services fonctionnant de façon concurrente et sécurisée dans différents systèmes d'exploitation. Plus récemment, la virtualisation est apparue dans le marché des ordinateurs de bureaux et permet d'exécuter sur une même machine physique différentes applications logicielles développées pour différents systèmes d'exploitation. La virtualisation a donc une utilisation évidente dans le domaine de la sécurité. En effet, une même machine peut alors héberger plusieurs systèmes d'exploitation indépendants, sans faire d'hypothèses sur leur fiabilité respective. La sécurité de chaque système d'exploitation vis-à-vis des autres est alors assurée par un logiciel de confiance : le système d'exploitation qui gère la machine physique.

2.4.1 Concepts

La virtualisation a introduit deux concepts : le concept de machine virtuelle et le concept d'hyperviseur. Une machine virtuelle est le conteneur dans lequel s'exécute une pile logicielle indépendante ; il s'agit typiquement d'un système d'exploitation, que l'on désigne alors comme système d'exploitation invité. L'hyperviseur est la couche logicielle de confiance qui permet l'exécution des différentes machines virtuelles en leur fournissant un environnement de travail. Popek et Goldberg [18] ont été les premiers à formaliser les conditions nécessaires pour le principe de l'hypervision. Ces conditions peuvent être résumées sous la forme des trois principes suivants :

1. Exécution équivalente : les programmes fonctionnant dans une machine virtuelle s'exécutent de façon identique à leur exécution native, à l'exception de différences d'ordre temporel ou de disponibilité des ressources.
2. Performance : toutes les instructions non critiques doivent être exécutées par le matériel directement, sans l'intervention de l'hyperviseur.
3. Sûreté : l'hyperviseur contrôle toutes les ressources matérielles.

La virtualisation s'appuie sur deux caractéristiques qu'un processeur doit avoir pour supporter la virtualisation :

- Le processeur doit offrir au moins deux niveaux de privilège. Les instructions privilégiées ne peuvent être exécutées que dans le mode le plus privilégié et sont piégées si elles le sont dans un mode moins privilégié.
- Toutes les instructions dites sensibles, c'est-à-dire qui ont accès en lecture ou écriture à des ressources de configuration du système, doivent être privilégiées.

Un tel processeur présentant des caractéristiques favorisant la virtualisation peut alors exécuter l'hyperviseur dans le mode le plus privilégié et les machines virtuelles dans un mode moins privilégié. Les machines virtuelles sont alors isolées entre elles puisque l'exécution d'une instruction sensible, susceptible de modifier l'état global de la machine, est systématiquement piégée par l'hyperviseur, et le plus souvent émulée.

En ce qui concerne les accès mémoires des machines virtuelles, ceux-ci doivent être relogés, c'est-à-dire qu'ils subissent une modification pour ainsi accéder seulement à l'espace mémoire alloué à la machine virtuelle. Ceci induit nécessairement l'utilisation d'une unité de mémoire virtuelle, pouvant ainsi réaliser cette relocalisation.

2.4.2 Types de virtualisation

Il existe deux méthodes principales de virtualisation, qui se différencient dans la façon dont les systèmes d'exploitation invités interagissent avec l'hyperviseur : la virtualisation complète et la paravirtualisation.

Dans le cas de la virtualisation complète, une machine virtuelle doit simuler suffisamment de matériel pour autoriser l'exécution d'un système d'exploitation invité non modifié. Dans ce cas, le système d'exploitation invité n'a pas conscience d'être exécuté dans une machine virtuelle et les instructions sensibles exécutées par celle-ci sont en fait piégées et émulées par l'hyperviseur. La virtualisation complète possède l'avantage de permettre l'exécution de systèmes d'exploitation non modifiés mais en contrepartie est plus complexe à mettre en œuvre. La virtualisation complète offre de meilleures garanties en termes d'isolation et de sécurité des machines virtuelles que la paravirtualisation, en plus d'offrir une portabilité totale d'un même système d'exploitation, puisque celui-ci peut être à la fois utilisé de façon native ou virtualisée.

Dans le cas de la paravirtualisation, un système d'exploitation invité a conscience d'être exécuté dans une machine virtuelle. Il interagit alors directement avec l'hyperviseur, qui lui propose un ensemble de services au travers d'une interface spéciale. Cette méthode implique une modification du système d'exploitation invité, qui doit alors être porté explicitement sur un hyperviseur paravirtualisé, ce qui n'est souvent pas réalisable pour les systèmes d'exploitation propriétaires dont le code source n'est pas disponible. La figure 2.8 montre l'approche de la paravirtualisation sur des architectures x86.

2.4.3 Virtualisation de matériel

La virtualisation nécessite de virtualiser le matériel, en particulier les ressources de type processeur et mémoire, ainsi que les périphériques.

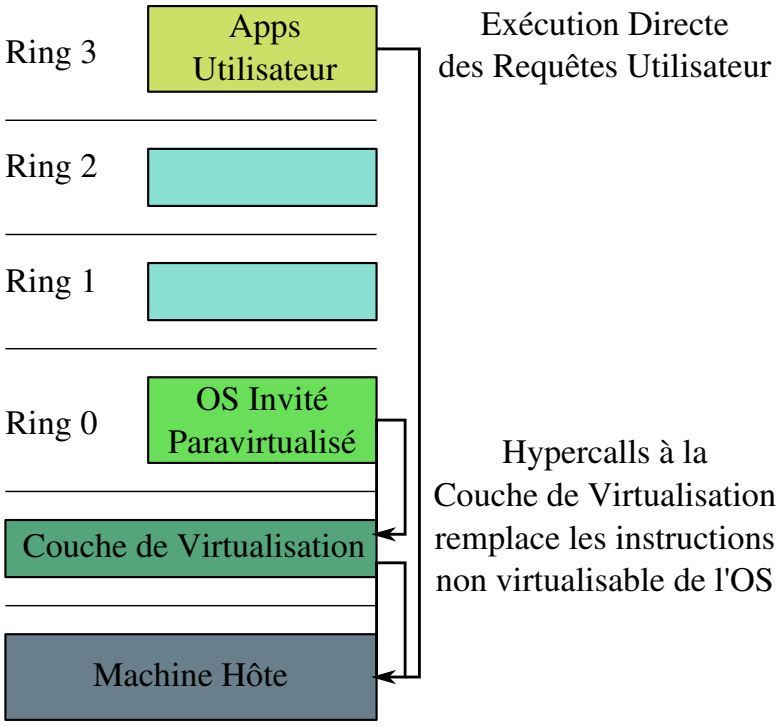

FIGURE 2.8 – Paravirtualisation sur des architectures x86
***Source** : D.Marshall [19]*

Virtualisation des processeurs

Lors d'une virtualisation complète, les instructions privilégiées exécutées par une machine virtuelle sont piégées puis émulées par l'agent logiciel de confiance. Lorsqu'une architecture comporte des instructions sensibles non privilégiées, celles-ci ne peuvent être automatiquement piégées par l'hyperviseur car elles peuvent être exécutées en mode utilisateur. Il est alors nécessaire que l'hyperviseur effectue une analyse des instructions appartenant à la machine virtuelle avant que celles-ci soient exécutées. Ceci peut être fait grâce à la technique de *binary translation*. Cette technique consiste à scanner le code que le noyau du système invité veut exécuter à un certain moment et de remplacer les instructions sensibles non privilégiées par des instructions privilégiées permettant ainsi à l'hyperviseur de piéger ces instructions. Cette analyse peut être faite soit de façon statique sur tout un programme ou alors de façon dynamique sur un bloc de base (un ensemble d'instructions se finissant par une instruction de branchement). En ce qui concerne la paravirtualisation, l'utilisation d'instructions sensibles est prohibée par la modification du code noyau du système invité. Ces instructions sont alors remplacées par des appels directs à l'hyperviseur.

Virtualisation de la mémoire

La virtualisation de la mémoire est très semblable au principe de mémoire virtuelle. Pour exécuter plusieurs machines virtuelles sur une unique machine il est nécessaire de rajouter un niveau d'adressage. L'hyperviseur va alors gérer le *mapping* de la mémoire physique de la machine virtuelle, appelée mémoire machine, sur la mémoire de la machine hôte, appelée mémoire physique. Le système invité va alors être en charge de gérer le *mapping* de la mémoire virtuelle sur la mémoire physique de la machine virtuelle (mémoire machine). La machine virtuelle ne possède pas d'accès direct à la mémoire physique. L'hyperviseur utilise des *shadow page tables* pour accélérer les traductions entre mémoire virtuelle et mémoire physique. Comme le montre la flèche rouge en pointillé dans la figure 2.9 l'hyperviseur utilise les TLBs pour faire le *mapping* entre les adresses virtuelles et leur correspondance en adresse physique, permettant ainsi d'éviter une double traduction à chaque accès. Lorsque une machine virtuelle modifie le *mapping* entre la mémoire virtuelle et sa mémoire machine, l'hyperviseur doit alors mettre à jour sa *shadow page table*.

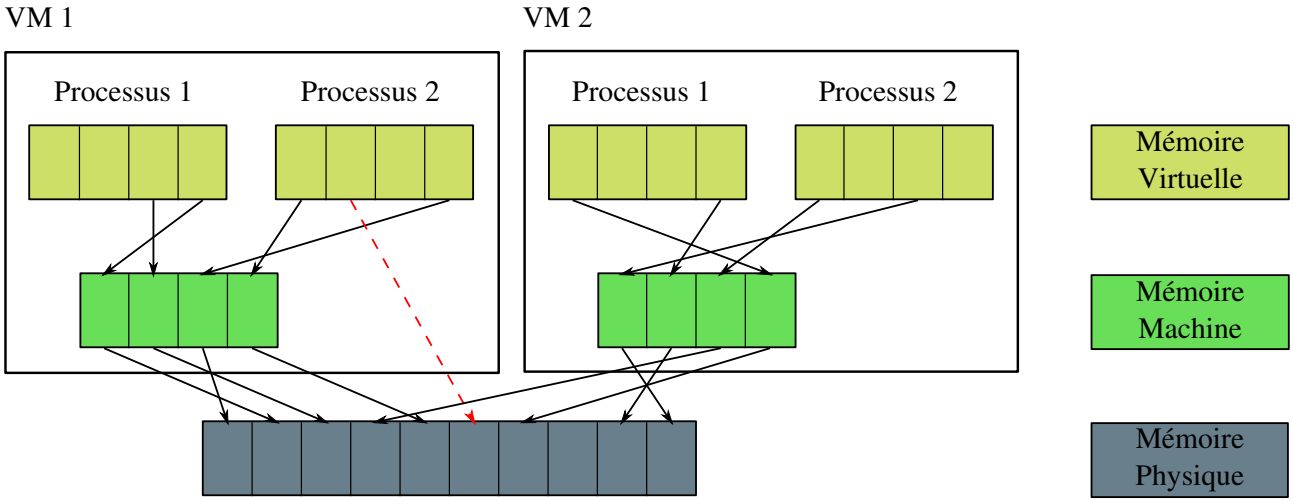

FIGURE 2.9 – Virtualisation de la mémoire
***Source** : D.Marshall [19]*

Virtualisation des périphériques

La virtualisation permet aussi de virtualiser les périphériques. En effet comme l'expliquent Abramson et al. [20], il existe principalement trois types de virtualisation en ce qui concerne les accès aux périphériques : l'*émulation*, la *paravirtualisation* et l'*assignation directe*.

L'émulation consiste à implémenter le matériel de façon strictement logicielle. Autrement dit, l'hyperviseur va venir proposer un modèle de périphérique à une machine virtuelle dont celle-ci possède déjà des pilotes [21]. La machine virtuelle va alors interagir avec ce modèle comme s'il s'agissait d'un vrai périphérique matériel. L'hyperviseur va alors proposer un modèle de périphérique dans

un certain espace d'adressage où chaque accès par la machine virtuelle sera piégé et émulé par l'hyperviseur. L'hyperviseur doit être capable d'envoyer des interruptions à la machine virtuelle lorsque le périphérique signale que celui-ci a terminé la requête. Traditionnellement, ce mécanisme est mis en place à l'aide d'un *Programmable Interrupt Controller* (PIC). L'hyperviseur doit alors aussi proposer un modèle de PIC pour permettre à la machine virtuelle d'y accéder, typiquement pour émuler l'acquittement de l'interruption du périphérique. L'émulation permet de ne pas modifier les pilotes des systèmes invités, et donc de permettre la virtualisation complète, mais présente un surcoût important en terme de performance car chaque interaction entre la machine virtuelle et un périphérique nécessite une transition par l'hyperviseur. Un autre désavantage de l'émulation provient du fait que les modèles de périphériques doivent être très précis pour refléter de façon stricte le comportement du matériel. La figure 2.10 illustre ce mécanisme d'émulation des périphériques.

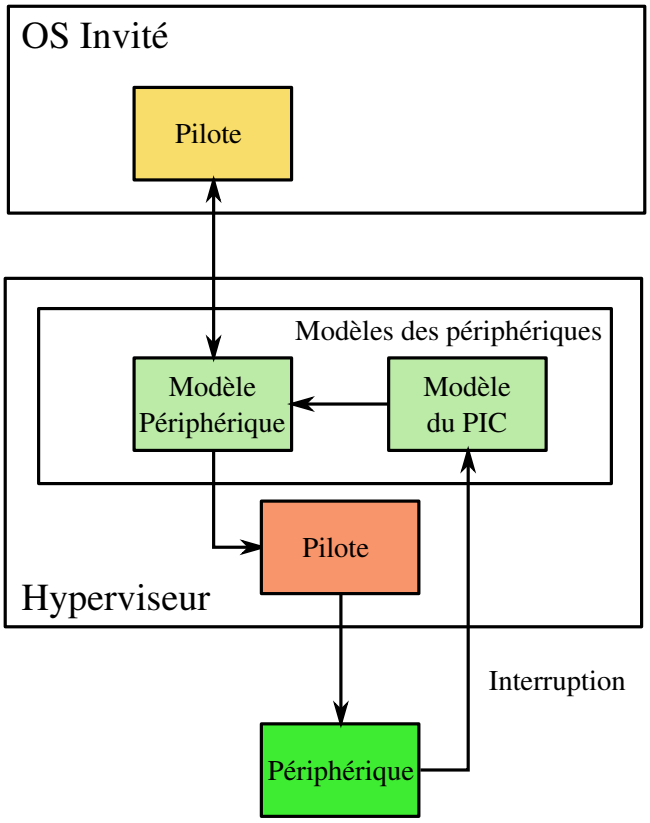

FIGURE 2.10 – Mécanisme d'émulation des périphériques
Source *: D.Abramson et al. [20]*

La paravirtualisation [22, 23] consiste à offrir au système invité une interface pour communiquer avec l'hyperviseur. Autrement dit, le système invité n'interagit plus directement avec les périphériques mais avec une interface d'entrée/sortie de l'hyperviseur. De ce fait, il est nécessaire de modifier à la fois les pilotes des systèmes invités mais aussi le mécanisme des interruptions car généralement celui-ci est remplacé par une technique basée sur des événements logiciels. La paravirtualisation des périphériques présente de bonnes performances comparée à l'émulation car elle permet de réduire le nombre d'interactions entre l'hyperviseur et le système invité, mais nécessite la modifications

des systèmes d'exploitations invités et donc ne permet pas la virtualisation complète. La figure 2.11 illustre le mécanisme de paravirtualisation dans le cas des accès d'entrée/sortie.

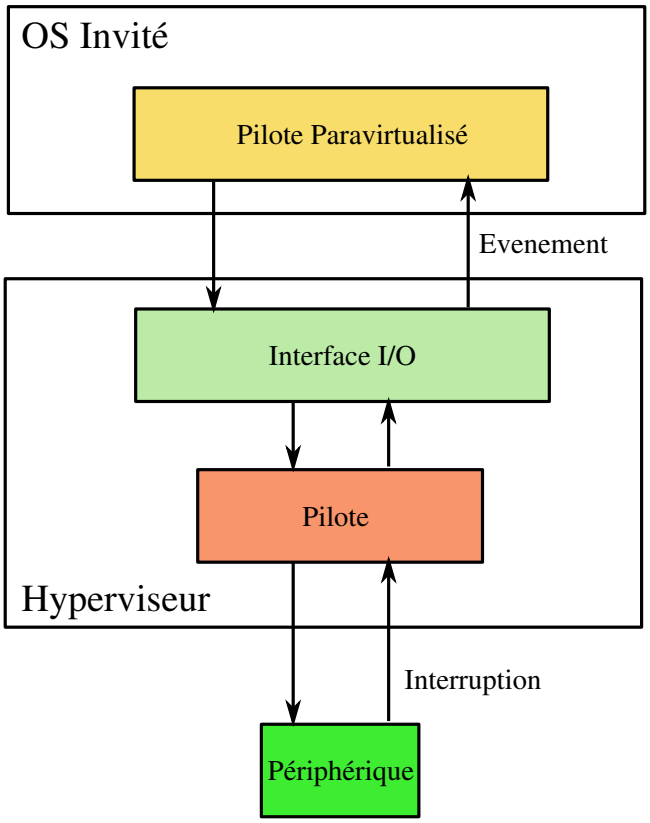

FIGURE 2.11 – Mécanisme de paravirtualisation des périphériques
Source *: D.Abramson et al. [20]*

L'assignation directe consiste à allouer un périphérique de façon exclusive à un système invité. En conséquence, celui-ci peut directement et nativement s'adresser au périphérique en utilisant son pilote existant. Néanmoins, l'hyperviseur doit quand même gérer la redirection des interruptions du périphérique vers le système d'exploitation invité ciblé par l'interruption. Cette technique permet en outre d'alléger l'hyperviseur car ce dernier n'a plus à fournir d'implémentation des pilotes des périphériques pour les systèmes d'exploitation invités. Il existe cependant un problème car les machines virtuelles s'exécutent dans un espace d'adressage virtuel qui ne correspond pas à celui que connait le périphérique, puisque celui-ci fonctionne en adressage physique. Cela est d'autant plus problématique lorsque le périphérique en question possède une capacité DMA, puisque les adresses fournies par le système d'exploitation invité sont virtuelles et que le périphérique les interprètera comme des adresses physiques. De plus, le système d'exploitation invité peut, par le biais des périphériques DMA, accéder à des zones de mémoire qui ne lui sont pas autorisées. Une solution à ce problème est que l'hyperviseur fournisse des pilotes intermédiaires pour ces périphériques, dit *pass-through*. Ces pilotes sont responsables d'intercepter toutes les interactions entre un système d'exploitation invité et le périphérique DMA. Ces pilotes réalisent alors la traduction des adresses virtuelles en adresses physiques, en vérifiant que celles-ci ciblent bien des zones de mémoire autorisées [24]. Les

pilotes *pass-through* sont spécifiques à chaque périphérique DMA car ils doivent décoder le format de la requête envoyée au périphérique. La technique d'assignation directe présente de bien meilleures performances que la technique d'émulation ou de paravirtualisation [25, 26], car le nombre d'interactions entre les systèmes d'exploitation invités et l'hyperviseur est très réduit.

La virtualisation hybride consiste à combiner les mécanismes de paravirtualisation et d'assignation directe. Une machine virtuelle (éventuellement plusieurs) est dédiée à la gestion des entrées/sorties. Cette machine virtuelle contrôle alors les périphériques physiques à l'aide de l'assignation directe, utilisant ainsi les pilotes existants, et fournit aux autres machines virtuelles un accès en mode paravirtualisation pour ces périphériques. Cette technique permet de réduire la complexité de l'hyperviseur et de rendre indépendante la gestion des périphériques pour l'hyperviseur et les machines virtuelles standards. La figure 2.12 illustre ce mécanisme.

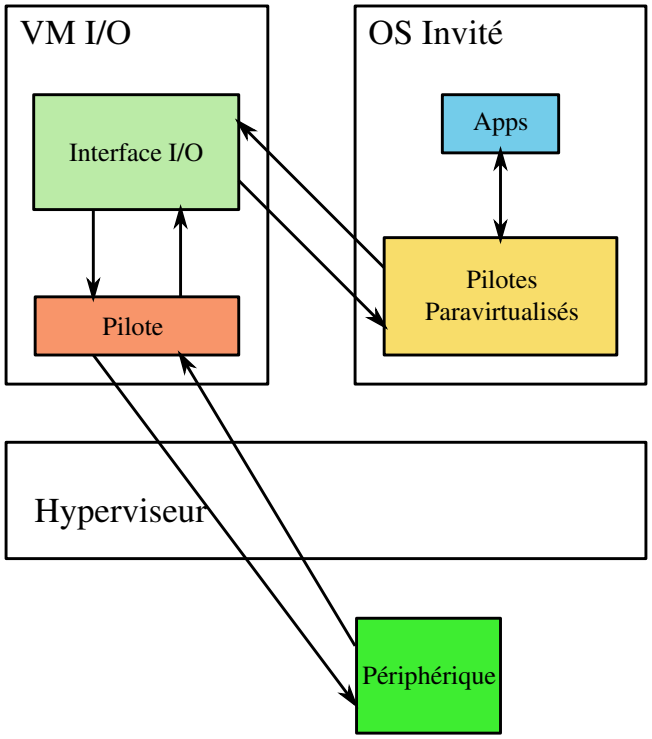

FIGURE 2.12 – Mécanisme de virtualisation hybride
Source *: D.Abramson et al. [20]*

2.5 Sécurité

Cette thèse vise à proposer une solution de sécurité permettant une cohabitation sécurisée de plusieurs machines virtuelles et d'un hyperviseur sur une même architecture manycore CC-NUMA. Cette solution se base sur la mise en place de mécanismes matériels et logiciels. Dans cette section, nous allons définir les propriétés de sécurité que nous cherchons à garantir.

2.5.1 Types de menace

La sécurité informatique repose sur trois propriétés : *la confidentialité, l'intégrité et la disponibilité*. Les interprétations de ces trois aspects varient, tout comme les contextes dans lesquels ils se présentent. L'interprétation d'un aspect dans un environnement donné est dictée par les besoins des individus, les coutumes et les lois.

La confidentialité [27, 28] est une propriété de sécurité qui permet de garantir qu'une information ne peut être accédée en lecture que par les entités autorisées. L'accès à une information privée peut être effectué de manière officielle et consentie entre deux entités, par un passage de message par exemple, ou alors de manière officieuse, par exemple une page en mémoire non effacée avant son allocation à une autre entité. Il est important de noter qu'une information fournie par une entité robuste à une autre entité non robuste met en danger la propriété de confidentialité : en effet, l'entité non robuste peut alors être victime d'une attaque permettant ainsi la divulgation de l'information. Une attaque visant la propriété de confidentialité peut compromettre l'intégrité d'une information, typiquement un détournement de mot de passe.

L'intégrité [29, 30, 31] est une propriété de sécurité garantissant qu'une information n'a pas été altérée par une entité non autorisée. Quelle que soit la nature de l'information, stockage à long terme comme une donnée sur un disque ou alors à court terme comme des données en mémoire vive, une attaque visant l'intégrité se traduit par un accès en écriture sur cette information. Les mécanismes permettant de garantir l'intégrité sont de deux types : les mécanismes de détection et les mécanismes de prévention. Les mécanismes de détection ne cherchent pas à prémunir d'une violation de l'intégrité d'une information mais signalent seulement que l'intégrité d'une donnée a été violée. Contrairement aux techniques de détection, les techniques de prévention permettent de maintenir l'intégrité des données en bloquant toutes les tentatives non autorisées de modification des données, ainsi que les tentatives de modification des données de manière non autorisée. La distinction de ces deux cas est importante. En effet, le premier consiste à une modification d'une donnée par une entité non autorisée alors que le deuxième fait référence à une modification d'une donnée par une entité autorisée mais dont la modification n'est pas autorisée. Par exemple un comptable modifiant les données bancaires, celui-ci étant autorisé à modifier ces données, pour cacher des détournements de fonds, cette modification n'étant pas autorisée. En mettant en place un système d'authentification et de contrôle d'accès, on peut aisément se prémunir du premier cas, contrairement au deuxième cas beaucoup moins aisé à contrer.

La disponibilité [32, 33] est une propriété de sécurité qui garantit que les entités autorisées peuvent accéder à un service dans un temps borné et considéré comme raisonnable. La disponibilité vise n'importe quel type de ressource, physique (processeur, mémoire) comme virtuelle (nombre maximal de processus, de fichiers ouverts). Une attaque typique visant la violation de la propriété de disponibilité, appelée attaque par déni de service, est la saturation du réseau : un grand nombre de machines

infectées envoie des requêtes au même serveur, celui-ci ne pouvant alors pas traiter les requêtes émanant d'utilisateurs non malicieux dans un temps acceptable. Un des principaux problèmes avec la disponibilité, est qu'il est compliqué de différencier une attaque par déni de service d'un pic brutal d'activité. Une attaque sur la disponibilité peut aussi mener à une seconde attaque portant cette fois sur l'intégrité ou la confidentialité.

Il est important de noter que les trois propriétés *confidentialité, intégrité et disponibilité* sont fortement liées entre elles : en effet, une attaque réussie portant sur une des trois propriétés peut aisément mener à la violation d'une autre propriété de sécurité.

2.5.2 Modèle de menace et Trusted Computing Base (TCB)

Dans cette thèse, nous faisons l'hypothèse d'avoir confiance dans le matériel. En particulier, nous considérons qu'il n'y a pas de *trojan*, et nous ne considérons pas les attaques physiques sur le matériel. Ainsi, l'intégralité de la plateforme matérielle fait partie de la *Trusted Computing Base*.

Le *Orange Book* [34] donne la définition suivante de la TCB : *"l'ensemble des mécanismes de protection, y compris le matériel, les firmwares et le logiciel, dont la combinaison est responsable d'assurer une politique de sécurité informatique"*. Autrement dit, la TCB est l'ensemble des éléments matériels et logiciels permettant de mettre en œuvre une base de confiance pour assurer une politique de sécurité.

Dans cette thèse, la TCB se compose donc principalement de l'ensemble de la plateforme matérielle et de quelques fonctions critiques de l'hyperviseur, typiquement les fonctions permettant de configurer le matériel et quelques mécanismes permettant de déployer ou d'arrêter les machines virtuelles.

Nous considérons deux types de menaces :
- Les attaques sur la confidentialité.
- Les attaques sur l'intégrité.

Nous considérons les systèmes d'exploitation s'exécutant sur la plateforme comme n'étant *pas de confiance* : en effet ceux-ci peuvent être corrompus ou contenir des failles permettant de corrompre l'ensemble de la plateforme, c'est pourquoi nous isolons les machines virtuelles entre elles.

Nous allons maintenant présenter le modèle de menace, dans lequel les menaces sont classifiées suivant trois paramètres : les menaces associées, les moyens nécessaires à mettre en œuvre pour réussir l'attaque et le niveau de risque de l'attaque. Pour cela, nous proposons une caractérisation des risques (2.1) suivant quatre niveaux : *très faible, faible, moyen et haut*. Pour caractériser le niveau de risque, nous utilisons trois paramètres : le coût de développement de l'attaque, les connaissances

techniques nécessaires et enfin le niveau nécessaire de connaissance du système pour que l'attaquant réussisse l'attaque. Comme le montre le tableau 2.1, plus l'attaque est facile à réaliser et plus le risque est élevé, mais il est important de séparer le risque et la conséquence d'une attaque car une attaque facile à réaliser peut néanmoins avoir de faibles répercussions sur la sécurité d'un système.

Risque	**Cout de l'attaque**	**Connaissance Technique**	**Connaissance du Système**
Très Faible	Prohibitif	Expert	Très Haute
Faible	Haut	Spécialiste	Haute
Moyen	Faible	Intermédiaire	Faible
Haut	Très Faible	Débutant	Aucun

TABLE 2.1 – Caractérisation à quatre niveaux du risque

Nous allons maintenant présenter l'impact de certains services de l'hyperviseur s'ils venaient à être corrompus. Les différents services que nous allons analyser sont : le *monitoring*, les services d'allocations des machines virtuelles, le contrôle des droits d'accès et les pilotes de périphériques. En ce qui concerne les machines virtuelles, nous adoptons une politique de sécurité garantissant l'intégrité et la confidentialité entre deux machines virtuelles mais nullement d'une machine virtuelle envers elle-même. Autrement dit, si des services du noyau du système d'exploitation d'une machine virtuelle venaient à être corrompus, ceux-ci ne pourraient donc pas venir corrompre les autres machines virtuelles présentes sur la plateforme mais pourraient empêcher le bon fonctionnement de cette même machine virtuelle.

Monitoring : les services de monitoring, s'ils venaient à être corrompus, pourraient mener à un dysfonctionnement en terme d'intégrité ou de confidentialité dans le cas de l'hyperviseur. Celui-ci pourrait avoir par exemple une mauvaise vision de l'état de sa plateforme et donc démarrer plusieurs machines virtuelles sur les mêmes ressources. Il faut donc pouvoir assurer que cela ne se produise pas.

Allocations : les services d'allocation de machines virtuelles (ainsi que ceux concernant leurs destructions) doivent être certifiés, car si ces services venaient à être corrompus, il serait alors impossible de garantir la moindre sécurité pour les machines virtuelles. Il est donc important de garder les codes concernant ces mécanismes les plus simples et petits possible (en termes de nombre de lignes de code), afin de minimiser au maximum la possible présence de fautes ou d'erreurs, et d'avoir le maximum de confiance en eux.

Contrôle des droits d'accès : l'hyperviseur étant l'entité possédant le plus de droits sur la plateforme, il est nécessaire de pouvoir assurer que celui-ci ne soit pas corrompu car cela pourrait mener à une perte d'intégrité ou de confidentialité pour les machines virtuelles. On peut néanmoins séparer l'hyperviseur en deux parties : une première considérée comme le noyau de l'hyperviseur, contenant l'ensemble des fonctions critiques de l'hyperviseur ; et une autre considérée comme la partie utilisateur contenant des fonctionnalité moins sensibles. La partie utilisateur ne doit pas pouvoir ve-

nir corrompre des données sensibles de l'hyperviseur (par exemple à l'aide de *buffer overflow*). Une séparation classique à l'aide de la mémoire virtuelle pourrait ainsi protéger les données sensibles de l'hyperviseur. En ce qui concerne la partie noyau de l'hyperviseur il est nécessaire de pouvoir restreindre son accès à la mémoire et aux périphériques alloués aux machines virtuelles. Des mécanismes matériels pourront être mis en place pour cantonner l'hyperviseur dans une zone de la plateforme.

Pilotes : toutes les fonctions configurant du matériel nécessitent d'être sûres, et cela est d'autant plus vrai pour les fonctions configurant le matériel utilisé lors du déploiement des machines virtuelles. Certaines mesures peuvent être prises en ce qui concerne les périphériques alloués aux machines virtuelles, en interdisant l'accès de l'hyperviseur aux canaux réservés aux machines virtuelles.

Le tableau 2.2 récapitule les différents scénarios expliqués ci-dessus.

Menace	**Élément corrompu**	**Attaque**	**Risque**
Déni de service	Monitoring	Indicateurs de ressources corrompus	Faible
Déni de service	Pilotes	Modification des requêtes	Moyen
Déni de service	Allocation	Refus d'allocation de machines virtuelles	Moyen
Intégrité et Confidentialité	Allocation	Mauvaise allocation d'une machine virtuelle	Haute
Intégrité et Confidentialité	Pilotes	Mauvaise configuration	Haute
Intégrité et Confidentialité	Droits d'accès	Écriture/Lecture de mémoire non autorisées	Haute
Fuite d'informations	Pilotes	Lecture de données latentes des registres	Moyen
Fuite d'informations	Allocation	Non remise à zéro des ressources allouées	Moyen

TABLE 2.2 – Attaques et menaces en provenance de l'hyperviseur

2.6 Problèmes identifiés

Dans cette thèse nous cherchons à tirer parti des nombreuses ressources de calcul et de mémoire des architectures manycore en déployant un grand nombre de machines virtuelles en parallèle. Pour cela nous décidons de ne pas faire confiance au logiciel, ou du moins nous essayons de réduire au maximum la surface d'attaque concernant le logiciel, et nous préférons modifier légèrement le matériel en ajoutant dans l'architecture des composants en charge de la sécurité des machines virtuelles et de l'hyperviseur. Nous souhaitons une isolation stricte, garantie par le matériel, entre les machines

virtuelles mais aussi avec l'hyperviseur. L'isolation stricte inter-VM et hyperviseur pose des problèmes en ce qui concerne l'accès aux périphériques, en effet les machines virtuelles doivent utiliser l'assignation directe car aucune communication entre les machines virtuelles et l'hyperviseur est possibles. Nous chercherons à implémenter un petit hyperviseur permettant ainsi de réduire sa surface d'attaque. Nous voulons pouvoir exécuter des systèmes d'exploitations non modifiés et donc utiliser la virtualisation complète, pour cela nous devons introduire un nouvel espace d'adressage. Nous ne souhaitons pas complexifier les cœurs de l'architecture ce qui nous contraint à réaliser la traduction de ce nouvel espace d'adressage en dehors des cœurs. Cela pose un problème pour la cohérence des caches car les caches L1 seront alors dans un espace d'adressage différent de celui des caches L2.

La suite de cette section vise à spécifier les différents problèmes soulevés par le contexte de la virtualisation sécurisée déployée sur des architectures manycore à mémoire partagée possédant un protocole de cohérence de cache matériel.

2.6.1 Cohabitation des machines virtuelles avec garanties d'intégrité et de confidentialité

De part leur nature, les architectures manycore permettent une utilisation massive de la virtualisation. En effet ces architectures possèdent beaucoup de ressources de calcul et de mémoire, ces ressources étant regroupées sous forme de clusters. Un déploiement de machine virtuelle sur différents clusters permettrait une autre utilisation des architectures manycore ; au lieu d'exécuter une application très fortement parallélisée, plusieurs piles logicielles indépendantes pourraient se partager la plateforme. Ces piles logicielles peuvent en plus fonctionner sous différents systèmes d'exploitation. Par ailleurs, dans un contexte de *cloud computing*, où différents utilisateurs viendraient exécuter leurs applications sur un serveur de calcul, l'aspect sécurité est primordial. La question de la cohabitation de plusieurs machines virtuelles sur des architectures manycore se pose alors : en effet, comment peut-on garantir que les machines virtuelles ne pourront pas gêner ou corrompre d'autres machines virtuelles s'exécutant sur la même plateforme ? De plus, il est important de pouvoir exécuter différents systèmes d'exploitation non modifiés pour éviter les problèmes de portabilité et de propriété intellectuelle. Dans cette thèse, nous chercherons une solution à la cohabitation des machines virtuelles sur des architectures manycore, en mettant en place des mécanismes logiciels et matériels permettant la virtualisation complète assistée par matériel, du fait que l'assistance matérielle offre de meilleures performances de virtualisation et une meilleure protection en terme de sécurité.

2.6.2 Limitation des droits de l'entité hyperviseur

Dans cette thèse, nous considérons que les hyperviseurs actuels deviennent de plus en plus complexes et lourds en termes de quantité de code. Ainsi, la probabilité qu'une faille existe augmente,

mettant alors en péril l'intégrité et la confidentialité des machines virtuelles s'exécutant sous son contrôle. Il devient alors nécessaire de limiter les droits de l'hyperviseur sur la machine, et plus particulièrement sur les zones où s'exécutent les machines virtuelles. Pour limiter l'hyperviseur, nous explorons le concept d'hyperviseur aveugle [35], en isolant l'hyperviseur sur des ressources dédiées et en l'interdisant d'effectuer des requêtes vers l'extérieur de sa zone allouée, au même titre que les machines virtuelles. Cependant, en limitant les droits d'accès à l'hyperviseur, comment peut-on assurer un bon fonctionnement de la virtualisation – autrement dit, comment un hyperviseur restreint peut-il, par exemple, démarrer ou arrêter des machines virtuelles ?

2.6.3 Dynamicité de création et de destruction des machines virtuelles

Dans cette thèse, nous nous imposons comme contrainte de gérer la dynamicité des machines virtuelles, c'est-à-dire de pouvoir démarrer puis arrêter des machines virtuelles en cours de la simulation de la plateforme, par exemple via des commandes d'un utilisateur. Dans le contexte énoncé précédemment de limitation des droits de l'hyperviseur, la question du démarrage des machines virtuelles devient primordiale, car l'hyperviseur n'est pas capable d'interagir avec l'extérieur, ou alors de manière très restreinte. De plus, nous ne faisons pas confiance à l'hyperviseur ni aux systèmes d'exploitation des machines virtuelles, c'est pourquoi nous explorerons un mécanisme d'isolation de l'intérieur, dans lequel les machines virtuelles s'isoleront par elle-même à l'aide d'une procédure logicielle assistée par le matériel. En ce qui concerne l'arrêt des machines virtuelles, comme l'hyperviseur ne peut pas interagir avec une machine virtuelle, nous avons mis en place un mécanisme matériel amenant à la destruction des machines virtuelles. Ce mécanisme n'est pas interruptible et conduit à la libération des ressources allouées à la machine virtuelle, qui peuvent ainsi être réutilisées par la suite pour une autre machine virtuelle. Il faut donc pour cela s'assurer que les ressources auparavant utilisées par la machine virtuelle à détruire sont dans un état vierge à la fin de la procédure d'extinction, notamment pour éviter des interblocages matériels.

2.7 Conclusion

Au cours de ce chapitre, nous avons vu le concept de virtualisation et ses utilisations. Par ailleurs, nous avons aussi noté l'émergence d'architectures possédant un grand nombre de ressources de calcul, pouvant ainsi être utilisées dans le contexte de la virtualisation. Cependant, un problème se pose au niveau de la sécurité car différents utilisateurs viennent exécuter des piles logicielles différentes, dont certaines peuvent être potentiellement dangereuses pour d'autres. Il convient alors de pouvoir garantir l'intégrité et la confidentialité des différentes machines virtuelles s'exécutant en parallèle sur la même machine physique. Au cours de cette thèse, nous allons répondre aux trois principales questions que nous avons relevées et qui sont exprimées ci-dessous :

1. *Comment faire cohabiter de façon sécurisée, en termes d'intégrité et de confidentialité, un large nombre de machines virtuelles s'exécutant sur un processeur manycore sans que les systèmes d'exploitation invités ne soient modifiés ?*
2. *Comment empêcher l'hyperviseur d'être omniscient et omnipotent sur l'ensemble de l'architecture ?*
3. *Comment assurer un déploiement et une destruction dynamiques et sécurisés des machines virtuelles ?*

Chapitre

3 État de l'art

Contents

Ce chapitre présente les travaux relatifs à l'hypervision et à la sécurisation d'environnements d'exécution. Dans la section 3.1, les différents types d'hyperviseurs existants seront présentés, et nous définirons les caractéristiques nécessaires à un hyperviseur pour répondre à notre problématique. Ensuite, nous étudierons dans la section 3.2 les extensions matérielles de virtualisation existantes, à savoir les solutions développées par Intel, AMD et ARM. Finalement, nous présenterons dans la section 3.3 différentes architectures sécurisées qui cherchent à répondre à la problématique d'exécution sécurisée de machines virtuelles.

3.1 Hyperviseurs

L'hyperviseur est un élément important de la virtualisation, car celui-ci est généralement utilisé pour gérer les différents systèmes d'exploitation invités s'exécutant sur une même plateforme [36, 37]. C'est un agent logiciel situé entre le matériel et les systèmes d'exploitation virtualisés, qui est en charge d'allouer les ressources matérielles aux systèmes d'exploitation invités. Par son rôle, l'hyperviseur est un élément critique en terme de sécurité puisque chaque faille présente dans l'hyperviseur peut mener à un ou plusieurs des points suivants :

- lectures non autorisées de données d'une machine virtuelle (violation de confidentialité) ;
- modifications non autorisées de données d'une machine virtuelle (violation d'intégrité) ;
- fuites d'informations – données présentes en mémoire ou dans les composants matériels qui peuvent être exploitées ensuite par une autre machine virtuelle malveillante.

De ce fait, l'hyperviseur doit être dans la *Trusted Computing Base*, i.e. dans les éléments de confiance au sein du système. C'est pourquoi il est primordial que les hyperviseurs demeurent aussi petits que possible, pour ainsi minimiser les risques d'être compromis. R. J. Masti et al. [38] définissent deux propriétés concernant la sensibilité de l'hyperviseur aux attaques : le fait d'avoir une faible empreinte et le fait d'avoir une interaction réduite avec les machines virtuelles. L'empreinte est généralement mesurée en termes de lignes de code (LoC). En effet, avoir moins de lignes de code signifie contenir moins de *bugs* en moyenne, et ainsi moins de possibilités pour un attaquant d'exploiter une faille. En utilisant des extensions matérielles dédiées à la virtualisation, les hyperviseurs peuvent atteindre des tailles de 4 000 LoC [39] tandis qu'un hyperviseur implémentant tous les mécanismes de virtualisation atteint des tailles de 100 000 LoC [40].

Les interactions entre l'hyperviseur et les systèmes d'exploitation invités ont lieu au démarrage et à l'arrêt d'une machine virtuelle, ainsi qu'à chaque fois qu'une machine virtuelle requiert un service depuis l'hyperviseur, comme par exemple lors d'un accès aux périphériques d'entrée/sortie. On parle de *principe de désengagement* de l'hyperviseur lorsque l'on réduit ses interactions avec les machines virtuelles au minimum, i.e. le démarrage et l'arrêt des machines virtuelles. Cela permet alors de réduire les possibilités pour un attaquant d'exploiter une faille dans les fonctions de l'hyperviseur.

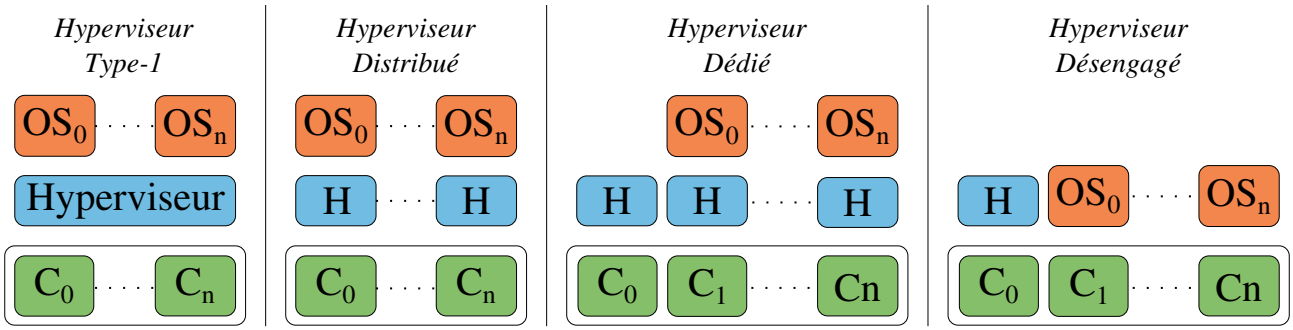

FIGURE 3.1 – Les différents types d'hyperviseurs
Source *: R.J.Masti et al. [38]*

Les hyperviseurs peuvent être classifiés en plusieurs catégories, présentées dans les sous-sections suivantes.

3.1.1 Hyperviseurs traditionnels

L'hyperviseur traditionnel, dit de type T1 (figure 3.1), est un hyperviseur composé d'une seule instance gérant intégralement toutes les ressources de la plateforme et interagissant fréquemment avec les machines virtuelles. Par exemple, chaque interruption d'entrée/sortie déclenche un changement de contexte vers l'hyperviseur. D'autres interactions peuvent être requises, en particulier pour la gestion de la mémoire s'il n'y a pas d'extension matérielle dédiée à la virtualisation, ce qui inclut un mode de privilège additionnel dans le cœur, combiné avec une extension de la MMU pour traduire les adresses machines en adresses physiques. De tels hyperviseurs nécessitent beaucoup de modifications matérielles pour réduire leurs interactions avec les machines virtuelles. Les hyperviseurs de type T1 sont les plus utilisés, et nous pouvons par exemple citer Xen [41] ou VMware [42]. Cependant, ils ne répondent pas aux deux critères évoqués précédemment, à savoir une faible empreinte et une faible interaction avec les machines virtuelles. En effet, même s'il est possible de réduire la taille des hyperviseurs en utilisant des extensions matérielles pour la virtualisation, la plupart de ces hyperviseurs possèdent 100 000 lignes de codes lorsqu'ils utilisent les techniques de para-virtualisation ou de *binary translation*. En ce qui concerne le principe de faible interaction, ces hyperviseurs, par conception, ne peuvent pas être satisfaisants.

3.1.2 Hyperviseurs distribués

Les hyperviseurs distribués consistent en plusieurs instances d'hyperviseurs où toutes sont en charge d'un sous-ensemble de l'architecture (figure 3.1). Comme pour les hyperviseurs de type T1, de tels hyperviseurs sont en charge du matériel et offrent une interface pour les accès aux périphériques. Ceci rend impossible une interaction minimale entre les machines virtuelles et l'hyperviseur.

Les hyperviseurs distribués utilisent les mêmes concepts que les systèmes d'exploitation dits *multi-kernel* [43, 44, 45]. Ce type de système d'exploitation vise à répondre aux problèmes de passage à l'échelle liés aux architectures comportant de plus en plus de cœurs. W. Shi *et al.* proposent MultiHype [46] comme hyperviseur distribué, dont les principaux avantages comparé aux hyperviseurs dits monolithiques sont le gain en performance dans le cas d'un déploiement sur une plateforme many-core (de part le concept de *multi-kernel*) et surtout des garanties de sécurité accrues. En effet, l'hyperviseur étant confiné dans plusieurs instances, une corruption d'une instance de l'hyperviseur peut difficilement en corrompre une autre, réduisant ainsi la surface d'attaque de la plateforme.

3.1.3 Hyperviseurs dédiés

Les hyperviseurs dédiés [47] s'exécutent sur des cœurs dédiés et gèrent les systèmes invités s'exécutant sur les autres cœurs. Ce type d'hyperviseur ouvre la voie à la suppression de la couche de virtualisation en laissant les machines virtuelles s'exécuter nativement sur les cœurs de la plateforme. Surtout, ce type d'hyperviseur peut respecter le principe de désengagement. En effet, un hyperviseur dédié n'est pas obligé de s'exécuter sur les mêmes cœurs que ceux de la machine virtuelle.

Hyperviseurs désengagés

Les hyperviseurs désengagés sont un type d'hyperviseurs dédiés qui limitent les interactions entre les machines virtuelles et l'hyperviseur pendant leurs exécutions, en permettant aux machine virtuelles d'être elles-mêmes en charge du matériel sur lequel elles s'exécutent. Cela permet d'augmenter la sécurité ainsi que la performance. Ces hyperviseurs peuvent à la fois avoir une faible empreinte et des interactions réduites avec les machines virtuelles, mais ils nécessitent généralement des extensions matérielles pour prendre en charge l'isolation des machines virtuelles. Par exemple NoHype [48] est un hyperviseur qui s'exécute sur un cœur dédié et qui ne partage pas de cœurs avec les machines virtuelles. Au démarrage d'une machine virtuelle, NoHype envoie un signal au cœur de démarrage de la nouvelle machine virtuelle, qui vient alors exécuter un *firmware* permettant la configuration de l'isolation de la machine virtuelle.

Hyperviseurs aveugles

Les hyperviseurs aveugles, présentés par P.Dubrulle *et al.* [35], sont un type d'hyperviseurs dédiés et désengagés qui cherchent à réduire la surface d'attaque de l'hyperviseur en lui interdisant d'accéder aux ressources matérielles allouées à une machine virtuelle une fois que celle-ci s'exécute. Les interactions entre l'hyperviseur et la machine virtuelle sont alors impossibles. L'hyperviseur peut

néanmoins arrêter la machine virtuelle, mais il ne peut pas accéder aux données de la machine virtuelle aussi bien pendant qu'après son exécution. Le concept d'hypervision en aveugle nécessite aussi des extensions matérielles.

3.1.4 Conclusion

Nous avons vu que les hyperviseurs sont en charge de la gestion des différentes machines virtuelles de la plateforme. Ils sont généralement responsables de la sécurité de la plateforme et leur rôle implique qu'ils peuvent être la cible d'attaques permettant de prendre le contrôle de l'intégralité de la plateforme. De plus, les hyperviseurs deviennent de plus en plus imposants en termes de lignes de code et donc cela induit nécessairement un nombre de failles grandissant. Plusieurs attaques aujourd'hui [49, 50, 51] poussent à croire que les hyperviseurs doivent être repensés et limités, permettant ainsi de réduire la surface d'attaque. Pour cela nous avons définit deux caractéristiques : faible empreinte et interactions réduites. Nous avons aussi présenté les différents types d'hyperviseurs existants. Seuls les hyperviseurs dédiés permettent de satisfaire ces deux caractéristiques. Nous pensons néanmoins qu'il est nécessaire de limiter au maximum les droits de l'hyperviseur, à savoir lui interdire l'accès aux ressources des machines virtuelles. C'est pourquoi nous explorerons dans notre travail le concept d'hyperviseur aveugle.

3.2 Support matériel pour la virtualisation sécurisée

Cette section vise à étudier les diverses extensions matérielles pour la virtualisation présentes dans les processeurs d'aujourd'hui. En effet, la virtualisation étant de plus en plus utilisée, les constructeurs de processeurs ont essayé de trouver des solutions matérielles permettant de réduire le surcoût induit par la virtualisation. Pour cela, nous présenterons les extensions permettant d'ajouter un nouveau niveau de privilège au sein du processeur. Nous présenterons ensuite les extensions matérielles permettant de gérer les *Shadow Page Tables* ainsi que la technologie Intel VT-d. Enfin, nous présenterons la solution ARM TrustZone.

3.2.1 Extension matérielle pour la virtualisation des processeurs : Intel VT-x et AMD-SVM

Intel et AMD ont respectivement développé deux technologies VT-x [52] et AMD-SVM [53] pour résoudre le problème lié à la virtualisation du processeur. Ces technologies permettent d'exécuter les systèmes d'exploitation invités dans les mêmes niveaux de privilèges que sur une exécution native. Pour cela, Intel et AMD ont ajouté un nouveau mode dans lequel l'hyperviseur peut s'exécuter. Ces

deux modes, nommés *VMX root* et *VMX non-root* pour la technologie Intel VT-x, possèdent les quatre niveaux de privilèges déjà existants. La figure 3.2 représente les quatre niveaux de privilèges et les deux nouveaux modes d'exécution. Ceci permet donc au système d'exploitation invité de s'exécuter dans les privilèges prévus et permet aussi à l'hyperviseur une certaine flexibilité dans l'utilisation des différents modes de privilèges, par exemple pour des services non critiques de l'hyperviseur.

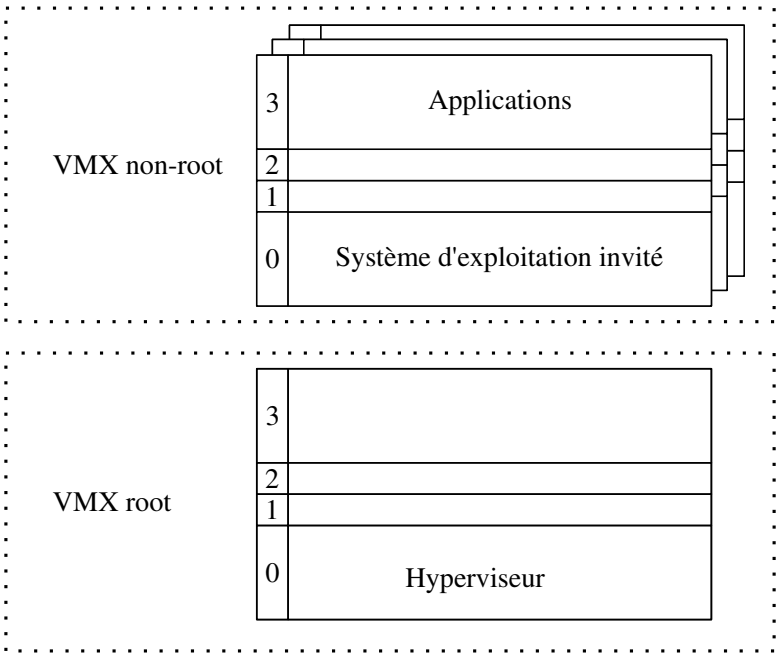

FIGURE 3.2 – Système utilisant la technologie Intel VT-x
***Source** : Rich Uhlig et al. [52]*

Intel VT-x, de même que AMD-SVM, définit deux nouvelles transitions entre les deux modes, par le biais de nouvelles instructions. Une transition est possible entre le *VMX root* et le *VMX non-root* et est appelée *VM entry*. Par exemple, cette transition est invoquée par l'utilisation des instructions `vmlaunch` et `vmresume`. La transition inverse, c'est-à-dire de la machine virtuelle vers l'hyperviseur, est aussi possible et est nommée *VM exit*.

Ces deux technologies introduisent une nouvelle structure de données : *Virtual-Machine Control Structure* (VMCS) pour Intel, et *Virtual-Machine Control Block* (VMCB) pour AMD. Cette structure permet de gérer les transitions *VM entries* et *VM exits* ainsi que le comportement du processeur lorsqu'il est en mode *VMX non-root*. Cette structure est divisée en deux sections : la zone de l'invité et la zone de l'hébergeur. Ces zones contiennent des informations relatives à l'état du processeur. Par exemple, un *VM entry* sauvegarde l'état du processeur dans la zone de l'hébergeur puis charge l'état du processeur sauvegardé dans la zone de l'invité alors qu'un *VM exit* réalise l'opération inverse.

Finalement, ces technologies permettent de pouvoir piéger plus aisément les instructions sensibles liées à la virtualisation et permettent au système d'exploitation invité de s'exécuter nativement sans que l'hyperviseur ne soit obligé de piéger toutes les instructions privilégiées du système invité.

Cela se traduit par une baisse du surcout lié à la virtualisation en réduisant le nombre d'appels à l'hyperviseur. Ces technologies nécessitent que l'hyperviseur et les machines virtuelles s'exécutent sur les mêmes cœurs pour pouvoir être fonctionnelles.

3.2.2 Extension matérielle pour la virtualisation de la mémoire : EPT et NP

Comme expliqué précédemment, pour renforcer l'isolation entre les différentes machines virtuelles, un hyperviseur peut utiliser un troisième espace mémoire. Cette technique oblige l'hyperviseur à maintenir une table des pages pour chaque machine virtuelle, permettant de traduire des adresses machine en adresses physiques. Cette gestion logicielle des défauts de pages induit un surcoût important en termes de performance. C'est pourquoi Intel et AMD ont développé des extensions matérielles permettant de résoudre ces défauts de pages. Ces technologies sont nommées *Extended Page Table* [54] pour Intel et *Nested Paging* [55] pour AMD.

En utilisant ces technologies, le système d'exploitation invité gère les *mappings* entre les pages virtuelles (VPN) et les pages physiques du système invité (MPN) dans sa table des pages, tandis que l'hyperviseur doit maintenir les traductions entre les pages physiques du système invité (MPN) et les pages physiques du système hôte (PPN). Ces dernières traductions sont situées dans un niveau de table des pages supplémentaire nommé *nested page tables*. Les tables des pages du système invité et les *nested page tables* sont utilisées par le matériel pour effectuer la traduction $VPN -> MPN -> PPN$. Quand une adresse virtuelle est émise, le matériel parcourt la table des pages du système invité, comme sur un système s'exécutant de façon native, et pour chaque MPN accédée, le matériel va alors parcourir en plus la *nested page table* pour déterminer la page physique du système hôte. Cette traduction supplémentaire permet d'éliminer le besoin de maintenir la cohérence en logiciel entre les tables de pages du système invité et celles du système hôte. Néanmoins, l'ajout de ce nouveau niveau de traduction augmente le temps nécessaire à la résolution d'une traduction d'adresse, en comparaison avec une exécution sans virtualisation, mais présente une hausse des performances de l'ordre de 40% par rapport à la version logicielle. La figure 3.3 montre le fonctionnement du mécanisme d'Intel *Extended Page Table*.

Le mécanisme de traduction doit assurer que toutes les adresses physiques obtenues pour une machine virtuelle ciblent uniquement une ressource allouée à la machine virtuelle, aussi bien de la mémoire que des périphériques. Traditionnellement, cette traduction est faite *via* une *Memory Management Unit* (MMU) à l'intérieur du premier niveau de cache [56, 57, 58], mais cela requiert que l'hyperviseur et la machine virtuelle partagent les cœurs.

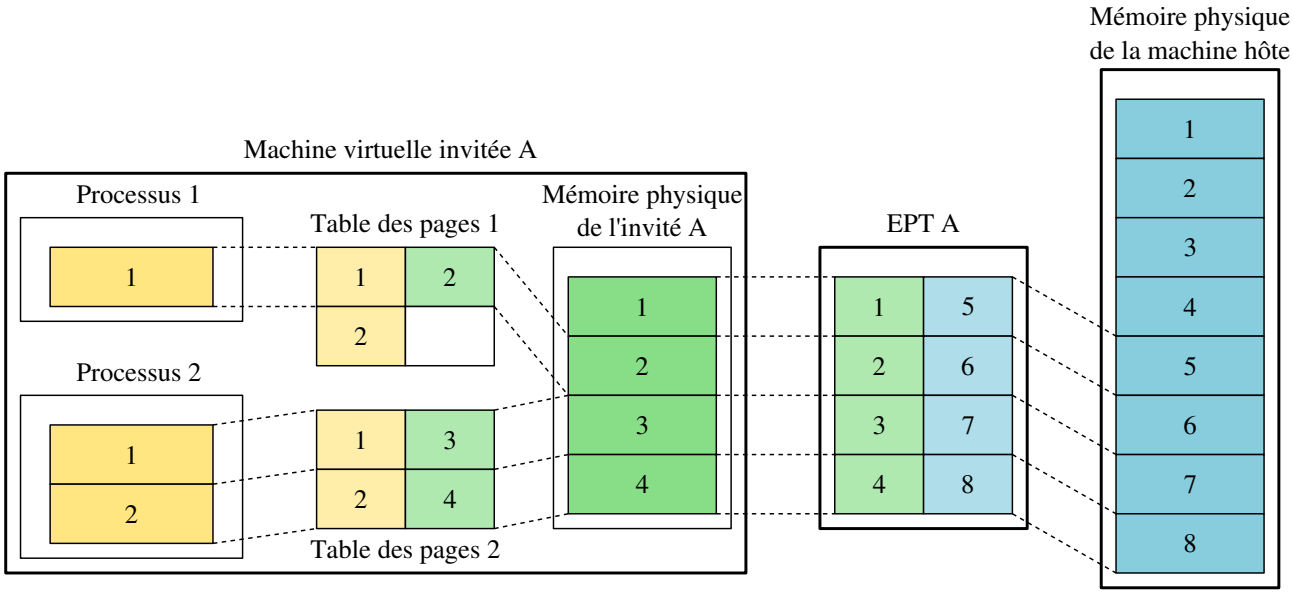

FIGURE 3.3 – Fonctionnement du mécanisme Extended Page Table
***Source** : Gil Neiger et al. [54]*

3.2.3 Extension pour la virtualisation des périphériques : Intel VT-d

Intel a développé la technologie VT-d dans le but d'améliorer l'isolation et les performances de la virtualisation. Cette technologie est basée sur la technique d'assignation directe des périphériques. Pour cela, Intel VT-d introduit un concept : le *DMA remapping*.

Intel VT-d introduit une IOMMU modifiée qui permet au logiciel de créer des domaines de protection pour les DMAs. Un domaine de protection est définit comme un environnement isolé auquel on a alloué un sous-ensemble de la mémoire de la machine hôte. Un domaine de protection DMA peut donc à la fois être de la mémoire allouée à une machine virtuelle ou à l'hyperviseur lui même. L'architecture VT-d permet d'assigner un ou plusieurs périphériques à un domaine de protection. Les restrictions d'accès d'un périphérique DMA sont réalisées en lui interdisant l'accès à la mémoire physique des domaines de protection qui ne lui sont pas assignés. Cela est réalisé par le biais de tables de traduction d'adresses. Les DMAs utilisent alors des adresses virtuelles (DVA) qui sont traduites par les tables de traduction en adresses physique du système hôte. Pour accélérer les traductions, Intel VT-d incorpore des IOTLBs permettant de stocker les traductions récentes. Le mécanisme de traduction est semblable à celui existant dans les caches L1.

3.2.4 ARM TrustZone

La technologie *TrustZone* [59], développée par ARM, consiste à distinguer deux mondes au sein d'une même plateforme matérielle : un sécurisé et un non sécurisé. Le monde non sécurisé peut typiquement héberger un système d'exploitation traditionnel, tandis que le monde sécurisé héberge une pile logicielle de confiance (hyperviseur), en charge du partitionnement de la plateforme (allocation de la mémoire et des périphériques). Cette technologie nécessite des extensions matérielles au niveau du processeur ainsi que l'ajout de modules matériels dans la plateforme.

Contrairement aux méthodes vues précédemment, où un nouveau niveau de privilège a été ajouté verticalement sous les autres, l'approche *TrustZone* duplique horizontalement les modes de privilège traditionnels. Un bit, nommé Non-secure Bit, permet de désigner dans lequel des deux mondes le processeur est à un instant donné. La MMU étant partagée entre les deux mondes, ses entrées sont étiquetées pour distinguer les deux utilisations. *TrustZone* autorise une virtualisation complète, c'est-à-dire qu'un système d'exploitation invité non modifié peut s'exécuter dans le monde non sécurisé de façon presque autonome. Les limites du système d'exploitation sont fixées par la pile logicielle s'exécutant dans le monde sécurisé.

La technologie *TrustZone* offre de plus un service de *Trusted Boot* permettant d'authentifier le code de la machine virtuelle lors du démarrage de celle-ci. Néanmoins, la technologie *TrustZone* ne permet pas d'héberger plusieurs machines virtuelles au sein d'une même plateforme matérielle.

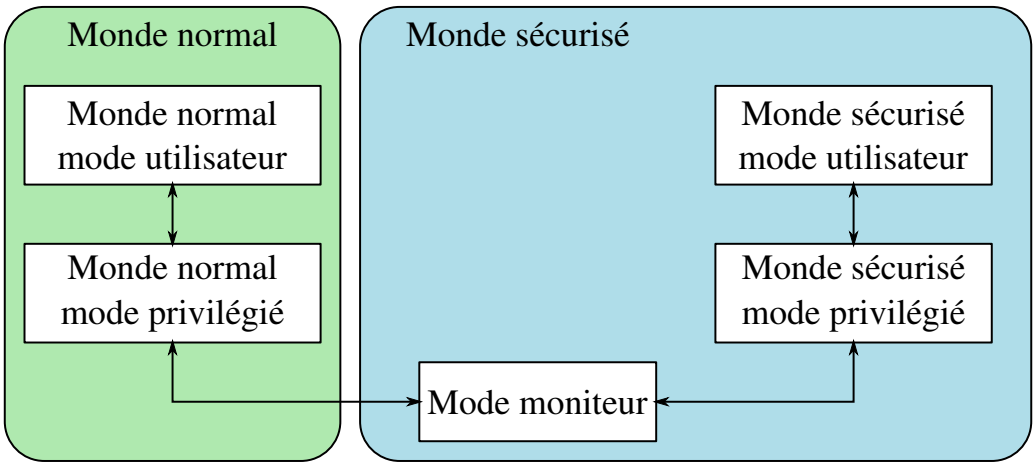

FIGURE 3.4 – Fonctionnement ARM Trustzone Technology

3.2.5 Conclusion

L'ensemble des solutions présentées dans cette section se base sur des extensions matérielles de virtualisation nécessitant le partage des cœurs entre l'hyperviseur et la machine virtuelle. Or, pour

un hyperviseur désengagé, et plus particulièrement pour un hyperviseur aveugle, il est impossible de faire cohabiter une machine virtuelle et l'hyperviseur sur les mêmes cœurs. De plus, ces extensions nécessitent la complexification des cœurs de calculs. Or, dans une architecture manycore, nous souhaitons garder les cœurs les plus simples possibles.

Concernant les mécanismes de traductions d'adresses machine en adresses physiques, une autre technique consiste à ajouter une MMU paginée entre le NoC et chaque initiateur sur celui-ci, au lieu d'implanter ce service au niveau des caches L1. Néanmoins, cette méthode présente quelques désavantages : premièrement, une MMU utilise généralement un cache de traduction (TLB) pour accélérer la traduction d'adresse. Cela implique un coût non négligeable en termes de matériel et de complexité pour gérer les TLB *misses*. Deuxièmement, l'hyperviseur doit créer une table des pages pour le *mapping* de la mémoire allouée à la machine virtuelle, et cette table des pages ne doit pas être accessible par les machines virtuelles ni par l'hyperviseur. Cette étape ne peut pas être faite intégralement en logiciel et nécessite l'introduction d'éléments matériels spécifiques. Troisièmement, une MMU est nécessairement plus lente pour effectuer la traduction à cause des *misses* dans les caches de traduction. Finalement, le principal avantage d'une MMU est la granularité de traduction, typiquement une page mémoire (e.g. 4Ko). L'avantage d'une granularité à la page peut être utile quand les machines virtuelles partagent des bancs mémoire, mais cela ne convient pas à notre hypothèse où nous voulons isoler physiquement les machines virtuelles.

Pour ces raisons, cette solution n'est pas entièrement satisfaisante et compte tenu du besoin en extension matérielle, une technique similaire à la segmentation a été investiguée dans cette thèse pour réaliser l'isolation physique de la mémoire [60].

3.3 Architectures sécurisées existantes

Dans cette section nous allons étudier diverses architectures existantes. Ces architectures visent à répondre à la problématique d'exécution de machines virtuelles, en leur fournissant des garanties d'intégrité et de confidentialité. Dans un premier temps, nous présenterons les différentes solutions qui ont été développées au Laboratoire d'Informatique de Paris VI (LIP6) et plus particulièrement les travaux de thèses de J. Porquet, G. Plouviez et S. Leroy. Dans un deuxième temps, nous étudierons les architectures Bastion, NoHype, HyperWall, H-SVM, SEMA et Intel-SGX.

3.3.1 Travaux antérieurs du LIP6

La thèse de Joël Porquet [61, 62] (thèse cifre ST soutenue en 2010) a visé le cloisonnement par un hyperviseur de machines virtuelles pour les processeurs ayant un faible nombre de cœurs, et donc

partagés dans le temps par plusieurs machines virtuelles. Pour cela, chaque machine virtuelle se voit attribuer une partie de l'espace d'adressage sous le contrôle d'une MPU paginée (*Memory Protection Unit*, appelée NoC-MPU). La NoC-MPU est placée entre les caches L1 et les caches L2. Chaque machine virtuelle dispose d'un numéro d'identification, et l'hyperviseur gère l'ordonnancement temporel des processeurs entre les machines virtuelles, ainsi que la configuration des NoC-MPU. L'identifiant de la machine virtuelle, configurable uniquement par l'hyperviseur, est ajouté à toutes les requêtes émanant de la machine virtuelle, ce qui permet aux NoC-MPU de vérifier les droits de la machine virtuelle qui émet la requête. Ce travail présente quelques limitations :

- les miss TLB des NoC-MPU utilisent un réseau ad hoc ;
- les NoC-MPU ne font pas de traduction d'adresse, ce qui impose une recompilation du code des machines virtuelles ;
- chaque machine virtuelle a ses propres périphériques, et ces derniers ne sont pas partagés ;
- le partage des processeurs empêche dans la pratique d'avoir des garanties sur l'absence de canaux cachés.

La thèse de Geoffrey Plouviez [63], soutenue en 2012, a visé le partitionnement d'un manycore pour plusieurs machines virtuelles en garantissant l'absence de canaux cachés. Dans ce travail, processeurs et mémoires sont dédiés à une machine virtuelle, mais les périphériques sont partagés sous le contrôle de l'hyperviseur. Chaque machine virtuelle dispose d'une partie propre de l'espace physique et les périphériques sont accessibles à des adresses physiques dont les requêtes sont interceptées et traduites par l'hyperviseur. L'hyperviseur a été écrit avec la méthode B pour prouver certaines propriétés de sûreté (allocation exclusive des ressources, cloisonnement des accès aux périphériques). Suite à ce travail, nous avons acquis deux convictions :

- l'hyperviseur ne doit pas intervenir pour les accès aux périphériques, car cela est trop couteux, et il est beaucoup plus efficace d'avoir autant de canaux que de machines virtuelles, et que ces canaux soient gérés directement par les machines virtuelles ;
- la méthode B est difficile à mettre en œuvre si l'on n'a pas les outils de génération automatique de code (le code a été produit à la main).

Au-delà de ces constats, la plateforme de validation utilisée par G. Plouviez ne gère pas la mémoire virtuelle ni la cohérence des caches.

Le travail effectué dans la thèse de Sylvain Leroy (thèse cifre Thalès, soutenance prochaine) a pour but d'exécuter plusieurs machines virtuelles sur un multicore (et non sur un manycore). Ce travail utilise, cette fois, une NoC-MMU paginée sur le modèle de la NoC-MPU, mais avec quelques différences. Les NoC-MMU gèrent la traduction d'adresses entre deux espaces d'adressage physique (hôte et invité), elles gèrent correctement les miss TLB, mais elles n'ont plus à gérer qu'un seul contexte par

cœur, car les cœurs ne sont plus partagés entre les machines virtuelles. Les périphériques sont désormais multi-canaux car chaque machine virtuelle dispose de ses propres canaux. Les NoC-MMU placées devant les canaux maîtres restent multi-contextes (multi-machine virtuelle). Ainsi, l'hyperviseur n'intervient plus que pour l'allocation de l'espace physique hôte aux machines virtuelles et il n'a plus besoin d'intervenir pendant l'exécution des machines virtuelles. Cela permet l'utilisation des périphériques par les machines virtuelles à leur débit maximum. Ce travail vise globalement le même but que celui de cette thèse, mais dispose d'hypothèses différentes, comme par exemple la non-dynamicité des espaces alloués, la restriction à un seul coeur des machines virtuelles, et le caractère omnipotent de l'hyperviseur, notamment en ce qui concerne la gestion des tables de page des NoC-MMUs.

En dehors des travaux réalisés au lip6, diverses propositions d'architectures sécurisées ont été faites au niveau industriel et académique. Nous présententons les principales propositions dans la suite.

3.3.2 Bastion

L'architecture Bastion [64], proposée par D. Champagne et R. B. Lee, met en place des mécanismes de protections logiciels et matériels pour offrir des garanties de sécurité pour l'exécution de piles logicielles critiques. Cette architecture nécessite des modifications au sein du microprocesseur et de l'hyperviseur en charge de la plateforme. Les mécanismes de protection mis en place dans Bastion ont pour but de sécuriser l'hyperviseur contre les attaques logicielles et matérielles. Leur modèle de menace comporte à la fois des attaques logicielles et matérielles. En ce qui concerne les attaques matérielles, ils réduisent leur zone de confiance au microprocesseur et proposent des solutions pour lutter contre des attaques par l'espionnage des bus, mémoires et des disques.

Le but de Bastion est de protéger l'exécution, ainsi que le stockage, de piles logicielles de confiance sur des piles logicielles qui ne sont pas de confiance. Premièrement, le microprocesseur de Bastion protège le stockage et la mémoire allouée à l'hyperviseur contre les attaques logicielles et matérielles. Deuxièmement, l'hyperviseur utilise des mécanismes de protection matérielle pour assurer l'exécution sécurisée de modules logiciels de confiance. La figure 3.5, adaptée depuis [64], présente un exemple d'utilisation de Bastion où trois applications (A, B, C) de confiance s'exécutent sur trois systèmes d'exploitation différents. Comme le montre la figure 3.5, cette solution fait pleinement confiance à l'hyperviseur et celui-ci partage les cœurs de calcul avec les machines virtuelles. D'ailleurs, les machines virtuelles s'exécutent toutes sur le même cœur, car cette solution ne propose qu'un modèle à base de monoprocesseur.

Bastion utilise des processeurs possédant des extensions de virtualisation, telles que les *Nested Page Tables*. L'hyperviseur est en charge de créer et de maintenir les tables de pages des machines vir-

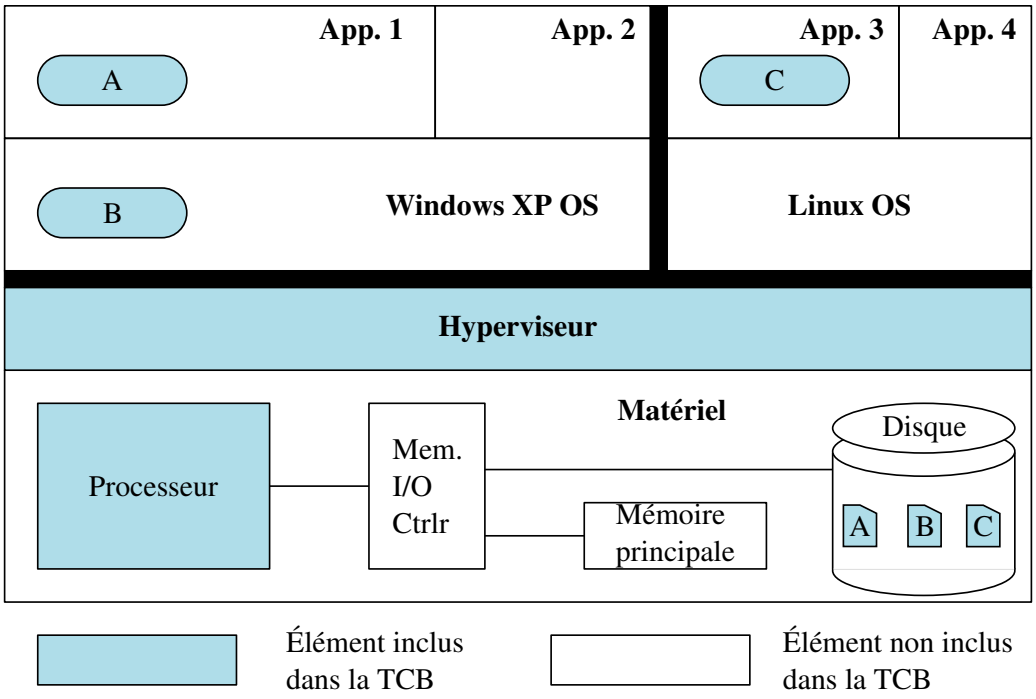

FIGURE 3.5 – Exemple d'utilisation de l'architecture Bastion
***Source** : D. Champagne et R. B. Lee [64]*

tuelles. Cette architecture propose un mécanisme de démarrage sécurisé pour l'hyperviseur se basant sur une méthode vérifiant l'intégrité de l'hyperviseur par le biais de *hash*. Le *hash* permet principalement de débloquer l'espace de stockage sécurisé. Chaque ligne de cache qui sort du cache L1 se retrouve chiffrée à l'aide d'un crypto-processeur intégré dans l'architecture, et est ensuite *hashée* pour pouvoir permettre une vérification de l'intégrité de la mémoire à l'aide d'un arbre de Merkle [65].

Une fois l'hyperviseur démarré, celui-ci peut lancer des machines virtuelles. Pour cela, il alloue pour chaque machine virtuelle un espace mémoire et copie le code de démarrage. Ensuite, il fait sauter le processeur à la séquence de démarrage de la machine virtuelle. Lorsque le système d'exploitation est démarré, celui-ci peut lancer des applications et celles-ci peuvent faire appel à une demande de création de zone sécurisée, permettant alors d'appeler une routine de l'hyperviseur qui a pour but de créer des zones de mémoires spécifiques pour l'application. La routine prend en paramètre un *pointer* sur un segment spécifique nommé *Security Segment*. Ce segment définit les pages du code et des données de l'application, ainsi que les points d'entrées autorisés et les interfaces de mémoire partagée autorisées. Le *Security Segment* doit être créé par le développeur de l'application avant l'utilisation de la routine, par exemple à la compilation de l'application. Le *Security Segment* a pour but de fixer les règles d'accès attribuées à la *Shadow Page Table* et contient trois éléments : le *hash* du code et des données de l'application, l'ensemble des pages de l'application et enfin les droits en lecture, écriture et exécution (R/W/X) pour ces pages.

Les évaluations présentées dans [64] montrent que cette solution ajoute un coût matériel d'envi-

ron 10% par rapport à l'architecture de référence et que la pile logicielle de confiance utilisée possède environ 50 000 lignes de codes.

L'architecture Bastion ne nous convient pas car elle nécessite d'avoir pleinement confiance dans l'hyperviseur et demande à l'utilisateur de fournir des informations pour le déploiement des applications par le biais des *Security Segment*. De plus, cette solution n'a, à notre connaissance, pas été déployée sur des architectures manycore.

3.3.3 NoHype

L'architecture NoHype [66], présentée par E. Keller *et al.*, vise à retirer entièrement la couche hyperviseur pour permettre de contrer des attaques sur celui-ci. De telles attaques peuvent mener à la corruption des machines virtuelles, comme le montre les expérimentations suivantes [67, 68]. Leur position est d'arrêter de protéger, ou de réduire la complexité de l'hyperviseur et de le retirer complètement. De plus, enlever la couche hyperviseur permet de réduire le surcoût lié à la virtualisation qui apparait lorsqu'une machine virtuelle fait appel à l'hyperviseur. Cette architecture possède trois points importants :

- Un coeur par machine virtuelle.
- Un partitionnement de la mémoire.
- Des périphériques virtuels dédiés.

Dans NoHype, l'hyperviseur est inexistant mais est remplacé par un logiciel en charge de gérer le déploiement et l'arrêt des machines virtuelles. Ce logiciel s'exécute sur les mêmes cœurs que ceux des machines virtuelles, mais le code de la machine virtuelle ne peut pas interagir avec le logiciel de gestion. Chaque cœur peut exécuter une seule et unique machine virtuelle, autrement dit NoHype ne permet pas que les cœurs soient partagés entre les différentes machines virtuelles. Cette stratégie permet d'éliminer les risques d'attaques par canaux cachés qui apparaissent lorsque des machines virtuelles partagent les ressources caches L1 [69, 70]. NoHype attribue à chaque machine virtuelle un espace d'adressage et interdit l'accès à celui des autres machines virtuelles. Pour cela, cette architecture utilise les extensions matérielles présentes dans les processeurs d'aujourd'hui, à savoir les *Extended Page Table* (EPT). Ces tables de pages sont construites au démarrage d'une nouvelle machine virtuelle, et permettent de *mapper* l'intégralité de la mémoire allouée à la machine virtuelle. Ainsi, le logiciel de gestion n'a pas à allouer dynamiquement de la mémoire pendant l'exécution de la machine virtuelle, permettant ainsi de réduire les interactions entre les machines virtuelles et le logiciel de gestion.

Le démarrage d'une machine virtuelle est réalisé par le logiciel de gestion qui envoie une IPI au cœur de démarrage (*Core Manager*) de la machine virtuelle. Celui-ci exécute alors du code de confiance qui a pour but d'allouer de la mémoire à la future machine virtuelle, en configurant les tables de pages. Ce code de confiance possède un accès aux ressources de la machine virtuelle et peut communiquer avec le logiciel de gestion par le biais des IPIs, par exemple lorsqu'une machine virtuelle désire s'arrêter. Le *Core Manager* attribue aussi à chaque machine virtuelle un accès direct aux périphériques. Pour cela, chaque machine virtuelle se voit allouer des périphériques virtuels, ce qui est réalisé à l'aide du *mapping* mémoire et l'utilisation du module IOMMU présent dans l'architecture. L'arrêt d'une machine virtuelle est également réalisé à l'aide du logiciel de gestion.

La figure 3.6 présente l'architecture NoHype dans son ensemble.

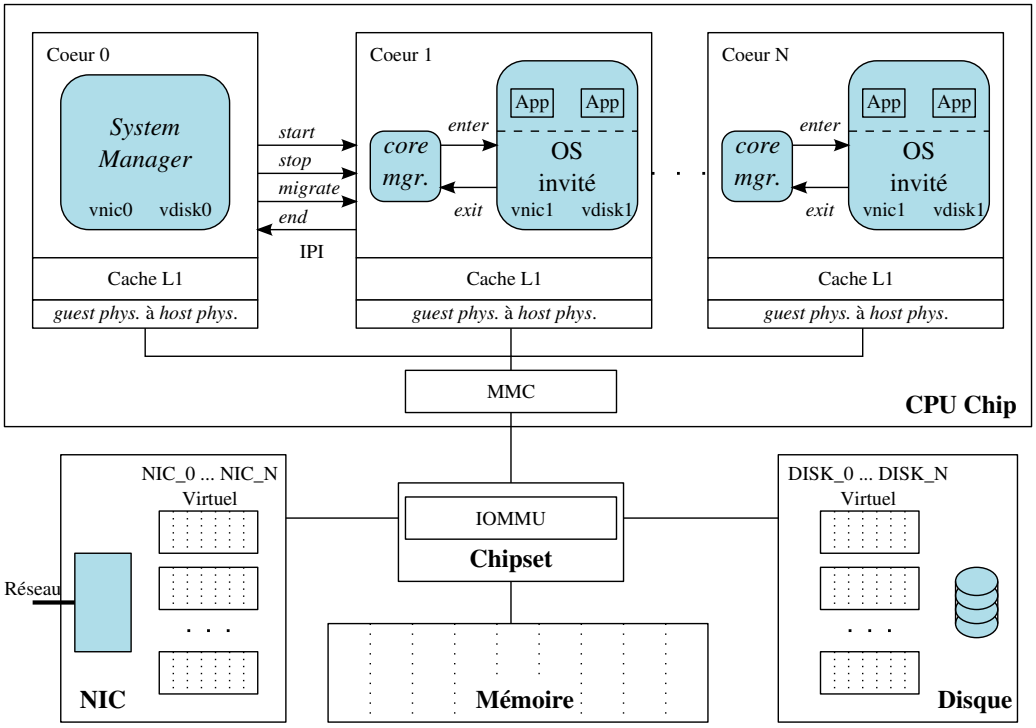

FIGURE 3.6 – Architecture du système NoHype
***Source** : E. Keller et al. [66]*

Dans l'article [48] les mêmes auteurs présentent leur prototype de l'architecture NoHype. Cette implémentation soulève plusieurs problèmes comme l'apparition d'un hyperviseur temporaire pour la phase de démarrage des systèmes d'exploitation des machines virtuelles ainsi que la nécessité d'apporter des modifications dans les systèmes d'exploitation invités. De plus, leur implémentation montre des baisses de performance assez importantes lorsque plusieurs machines virtuelles s'exécutent en parallèle. Pour ces raisons, l'architecture NoHype ne nous convient pas. De plus, cette solution n'a pas été déployée sur des architectures manycore.

3.3.4 HyperWall

L'architecture HyperWall [71], présentée par J. Szefer et R. B. Lee, vise à protéger les machines virtuelles d'un hyperviseur malicieux. Pour cela, les auteurs proposent des modifications matérielles au niveau des cœurs permettant de restreindre les accès de l'hyperviseur à la mémoire allouée à la machine virtuelle. L'isolation, ainsi que la restriction des pages de la machine virtuelle se fait à l'aide d'une table nommée *Confidentiality and Integrity Protection* (CIP). Cette table est uniquement accessible par le matériel et est stockée dans une zone de mémoire réservée, autrement dit une partie de la mémoire de la plateforme est rendue inaccessible pour le logiciel. La taille de cette table varie en fonction de la mémoire totale disponible dans la plateforme. Le modèle de menace utilisé dans l'architecture HyperWall ne prend pas en compte les attaques physiques mais uniquement les attaques logicielles. Tous les composants matériels font partie de la TCB. La sécurité des machines virtuelles n'est pas garantie en interne, autrement dit une application lancée dans une machine virtuelle peut attaquer une autre application de cette même machine virtuelle. Par ailleurs, les attaques en déni de service ne sont pas étudiées.

Cette solution demande l'ajout de nouvelles instructions au sein du processeur, rendant ainsi moins portable l'architecture car elle demande la modification de chaque cœur pouvant être utilisée dans l'architecture. De plus, dans cette architecture, l'hyperviseur et les machines virtuelles se partagent les cœurs.

La table CIP opère à la granularité page et est accédée par une MMU modifiée au sein de chaque cœur. A chaque accès mémoire, la table CIP est accédée pour vérifier que l'entité, en cours d'exécution, possède les droits d'accès sur cette page. Chaque page de la plateforme possède une entrée dans la table CIP, et cette entrée contient les règles de protection pour cette page. Il existe cinq valeurs possibles pour une entrée de la table CIP : la page n'est pas assignée, la page est assignée et accessible par l'hyperviseur et les DMAs, la page est assignée et non accessible par l'hyperviseur, la page est assignée et non accessible par les DMAs, et enfin la page est assignée et non accessible par l'hyperviseur ou les DMAs. Concernant la protection contre les DMAs, les modifications nécessaires pour la vérification de la table CIP sont aussi aussi réalisées au sein de l'IOMMU, lui permettant ainsi d'y accéder.

L'architecture HyperWall met aussi en œuvre un chiffrement, et un *hashage*, des registres généraux du cœur à chaque changement de contexte entre les machines virtuelles ou l'hyperviseur. Ceci permet de garantir l'intégrité de la machine virtuelle. Les chiffrés sont stockés en mémoire et accessibles uniquement par le matériel. Le *hash* des registres est lui stocké dans un nouveau registre du cœur.

HyperWall ajoute deux nouvelles instructions non privilégiées, `sign_bytes` et `trng`. L'instruction `sign_bytes` est utilisée pour signer les données tandis que l'instruction `trng` permet de faire appel au module *True Random Number Generator* (TRNG) matériel permettant la génération d'une nombre aléa-

toire. Trois instructions sont aussi modifiées : `vmlaunch`, `vmresume` et `vmterminate`. Ces instructions privilégiées sont utilisées pour démarrer, reprendre l'exécution et arrêter une machine virtuelle. Elles sont déjà présentes dans les processeurs modernes x86 possédant des extensions de virtualisation. L'instruction `vmlaunch` démarre une nouvelle machine et permet d'activer les protections d'HyperWall pour cette machine virtuelle. L'instruction `vmterminate` permet d'arrêter une machine virtuelle et plus particulièrement d'effacer la mémoire réservée à la machine virtuelle ainsi que les entrées de la table CIP correspondantes aux pages allouées. L'instruction `vmresume` permet d'effectuer un changement de contexte entre l'hyperviseur et la machine virtuelle. Elle réalise également une vérification de l'intégrité des registres d'HyperWall.

La figure 3.7 présente les différentes modifications matérielles nécessaires à l'implémentation de l'architecture HyperWall.

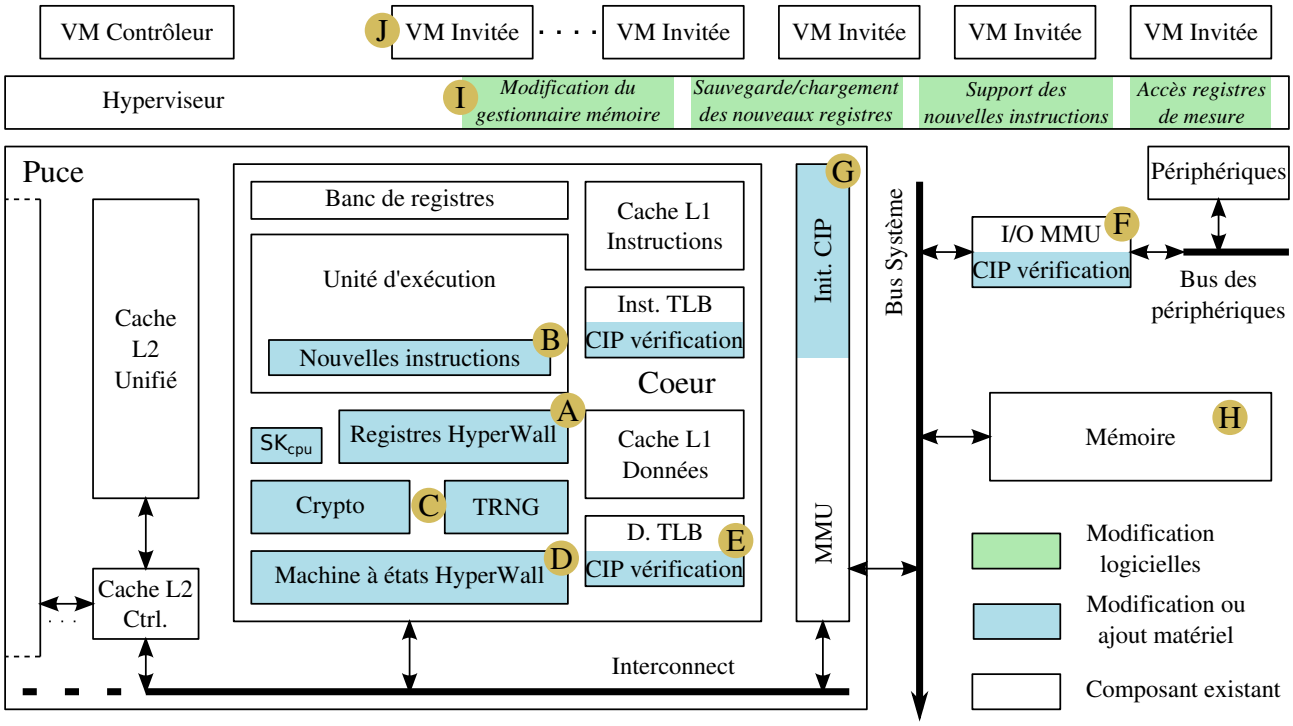

FIGURE 3.7 – Architecture HyperWall
***Source** : J. Szefer et R. B. Lee [71]*

De nouveaux registres (A) sont ajoutés permettant de stocker les informations de protection de la machine virtuelle en cours d'exécution. (B) correspond aux instructions ajoutées (`sign_bytes` et `trng`) et modifiées (`vmlaunch`, `vmresume` et `vmterminate`). Un module cryptographique (C) est nécessaire pour réaliser les opérations de chiffrement des registres. Le cœur d'HyperWall est la machine d'état (D) qui est responsable de la mise à jour de la table CIP lorsqu'une machine virtuelle est créée ou détruite. La logique des TLB est améliorée (E) pour prendre en compte les accès à la table CIP avant d'insérer de nouvelles traductions dans les TLB. Les vérifications d'accès sont réalisées en même temps que les traductions. Seules les traductions manquantes subissent une vérification des

CIPs. L'IOMMU nécessite aussi les mêmes modifications (F). La MMU (G) des cœurs est aussi modifiée, avec l'ajout d'un registre permettant de marquer le début de la zone de mémoire réservée au matériel. Une portion des DRAM (H) est réutilisée pour le stockage de la table CIP. L'hyperviseur (I) est modifié pour pouvoir exécuter les nouvelles fonctionnalités et faire appel aux instructions ajoutées. Aucune modification n'est nécessaire au sein des systèmes d'exploitation invités (J).

L'ensemble de ces modifications ajoute 400 octets de mémoire par cœur ainsi qu'une zone de mémoire réservée au stockage de la table CIP. Cette zone de mémoire est d'environ de 4 bits par page disponible dans la plateforme.

Cette solution ne nous convient pas car elle demande la complexification des cœurs, qui dans un contexte manycore peut présenter des problèmes de passage à l'échelle, et de plus l'ajout de nouvelles instructions rend l'architecture spécifique à un type de cœur (par exemple MIPS, ARM, Intel). De plus, bien que l'hyperviseur se voit restreint l'accès des pages mémoires des machines virtuelles, celui-ci partage toujours les cœurs avec les machines virtuelles. Enfin, cette solution n'a pas été appliquée dans le cadre des architectures manycore, et les évaluations ont été faites sur une architecture à 2 cœurs.

3.3.5 H-SVM

L'architecture sécurisée nommée *Hardware-assisted Secure Virtual Machine* (H-SVM) [56, 72], présentée par S. Jin *et al.*, vise à protéger la mémoire des machines virtuelles contre un hyperviseur malicieux. Pour cela, les auteurs proposent une solution matérielle et plus particulièrement des modifications au niveau des cœurs. Leur modèle de menace ne prend en compte que les attaques logicielles et l'intégralité du matériel est dans la TCB. Dans cette architecture, les machines virtuelles et l'hyperviseur partagent les cœurs de calcul et la mémoire, bien que certaines pages ne soient accessibles que par les machines virtuelles ou l'hyperviseur.

Le mécanisme de protection de la mémoire des machines virtuelles permet d'interdire la modification des tables de pages physiques des machines virtuelles par l'hyperviseur. Pour cela, l'hyperviseur doit faire appel au matériel, par le biais de nouvelles instructions, lorsqu'il veut allouer une nouvelle page, ou en désallouer. Le matériel est alors en charge de vérifier que la requête de l'hyperviseur est correcte. Par exemple, dans le cas d'une allocation de page, le matériel vérifie que la nouvelle page à allouer n'est pas déjà allouée à une machine virtuelle. Pour se protéger des accès DMA, l'IOMMU est modifiée et l'accès par l'hyperviseur aux tables de pages des I/O est interdit, seul le matériel peut y accéder.

Les tables de pages sont protégées en étant stockées dans un espace mémoire réservé et accessible uniquement par le matériel. Cette zone de mémoire contient plusieurs informations : le *VM Control*

Information, les *Nested Page Tables* et la *Page Ownership Table*. Le *VM control information* sert à enregistrer diverses informations comme la zone de mémoire utilisée pour sauvegarder les registres de la machine virtuelle lors des changements de contexte, l'adresse de la *Nested Page Table* de la machine virtuelle et la clé de chiffrement de la machine virtuelle qui est utilisée dans les opérations de *swapping*. Il existe autant de *Nested Page Tables* que de machines virtuelles, plus une qui est réservée pour l'hyperviseur. La table *Page Ownership* contient l'information d'appartenance d'une page. Cette table contient autant d'entrées que de pages dans la plateforme et chaque entrée permet au H-SVM de connaitre à qui appartient la page. Trois types de valeurs sont possibles : VMID (qui est l'identifiant unique attribué à une machine virtuelle), HYP ou H-SVM.

L'architecture H-SVM nécessite le support par le processeur de sept nouvelles instructions : `create`, `delete`, `switch`, `map`, `unmap`, `swap` et `set-share` permettant de faire appel au matériel et de réaliser les modifications nécessaires dans les *Nested Page Tables*. L'instruction `create` permet à l'hyperviseur de signaler au H-SVM que celui-ci veut démarrer une nouvelle machine virtuelle. Le matériel va alors créer une nouvelle *Nested Page Table* pour la nouvelle machine virtuelle, et initialiser la structure *VM Control Information* de la machine virtuelle. Le H-SVM retourne un identifiant unique de machine virtuelle à l'hyperviseur une fois ces initialisations terminées. L'instruction `delete` permet d'arrêter une machine virtuelle. Lorsque l'hyperviseur exécute cette instruction, H-SVM va alors effacer la mémoire utilisée par la machine virtuelle puis détruire la *Nested Page Table*, ainsi que la structure *VM Control Information* associée à la machine virtuelle. L'instruction `map` permet à l'hyperviseur de demander au H-SVM l'ajout d'une nouvelle page physique à la machine virtuelle, et donc de modifier la *Nested Page Table* en conséquence. Le H-SVM vérifie alors si la page à ajouter n'est pas déjà utilisée par une autre machine virtuelle. L'instruction `unmap` permet à l'hyperviseur de désallouer une page physique d'une machine virtuelle : le H-SVM efface alors le contenu de cette page et modifie la *Nested Page Table* en conséquence. L'instruction `switch` permet d'effectuer un changement de contexte entre l'hyperviseur et la machine virtuelle. Le H-SVM doit alors effectuer une sauvegarde des registres de la machine virtuelle. L'instruction `swap` permet à l'hyperviseur, ou au système d'exploitation invité, d'évincer des pages physiques pour les stocker dans le disque. Enfin, l'instruction `set-share` permet à un système d'exploitation invité de déclarer des pages physiques partagées avec l'hyperviseur ou d'autres machines virtuelles. Ces deux dernières instructions nécessitent la modification des systèmes d'exploitation invités.

La figure 3.8 présente l'architecture H-SVM. La figure 3.8 ne présente pas l'instruction `delete` mais leur article [72] montre que celle-ci existe.

Cette solution est vulnérable à la désallocation, et donc à la remise à zéro, d'une page utilisée par une machine virtuelle. En effet, l'hyperviseur peut désallouer une page, et la ré-allouer instantanément à la machine virtuelle. La machine virtuelle peut alors utiliser cette page sans savoir qu'elle a été remise à zéro.

Les auteurs présentent dans [72] une implémentation de H-SVM sur un processeur multicœur.

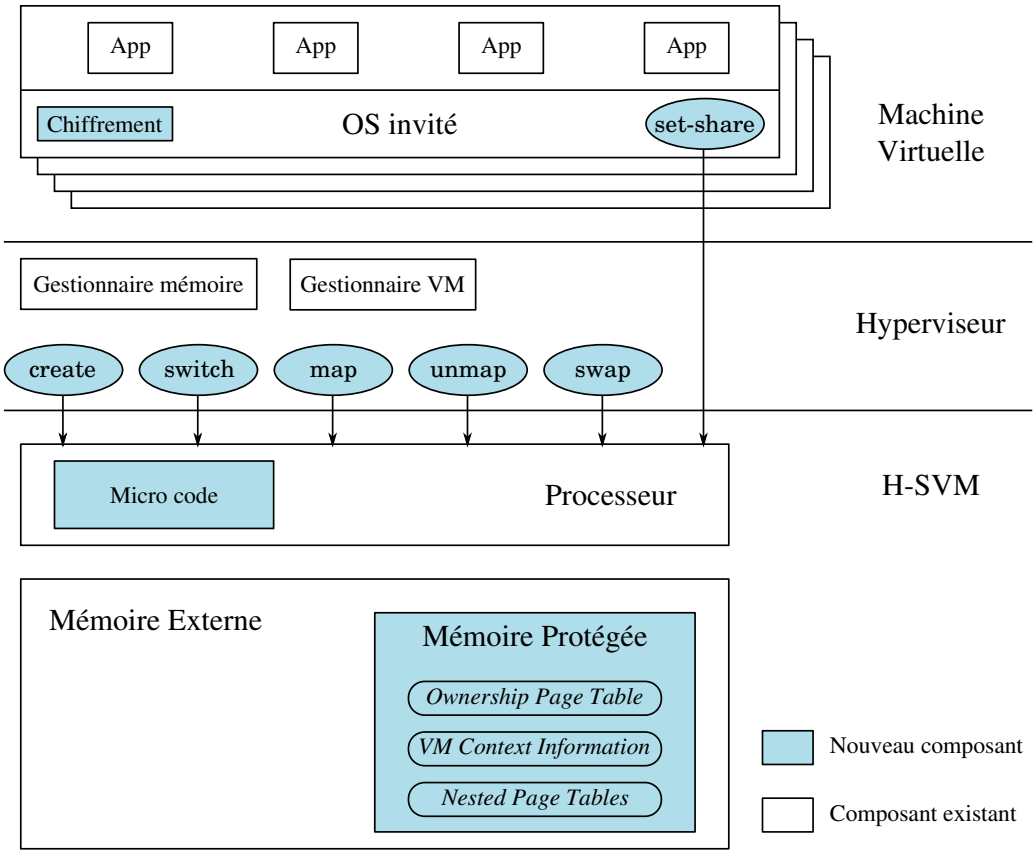

FIGURE 3.8 – Architecture du système H-SVM
***Source** : S. Jin et al. [56, 72]*

Dans leur implémentation, le code nécessaire pour implémenter de façon logicielle H-SVM est de 1 500 lignes. De plus, il apparait deux manques en terme de sécurité. Premièrement, les registres des machines virtuelles ne sont pas protégés et peuvent donc être modifiés par l'hyperviseur ; deuxièmement le registre pointant sur l'adresse de la *Nested Page Table* est accessible par l'hyperviseur. Celui-ci peut alors modifier le registre et donc forcer une machine virtuelle à utiliser une fausse table des pages.

L'architecture H-SVM ressemble fortement à l'architecture HyperWall et pour les mêmes raisons, à savoir l'ajout d'instructions et le partage des cœurs, cette solution ne nous convient pas. De plus, l'hyperviseur peut désallouer des pages utilisées par une machine virtuelle sans lui demander la permission et cela présente donc une vulnérabilité aux attaques par *re-mapping*. Enfin, cette solution n'a pas été appliquée au cadre des architectures manycore.

3.3.6 SEMA

L'architecture SEMA [38], proposée par R. J. Masti *et al.*, vise la sécurisation des plateformes manycore. Pour cela, les auteurs proposent d'utiliser le principe de partition logique [73], qui est un sous-ensemble de la plateforme permettant d'exécuter des piles logicielles indépendantes, sur des architectures manycore. Les partitions logiques sont utilisées dans les systèmes de sécurité pour isoler des parties de systèmes d'exploitation vérifiés formellement [74, 75] ou simplement dans le déploiement d'hyperviseur commerciaux [76, 77]. Par exemple, IBM utilise cette technique dans leur Cloud *Infrastructure as a Service* (IaaS) [78]. Cette architecture permet l'utilisation de la technique de désengagement de l'hyperviseur ce qui offre une réduction de la surface d'attaque, car les interactions entre les machines virtuelles et l'hyperviseur sont minimales.

L'architecture manycore choisie pour leur solution est le processeur Intel SCC [14]. Celle-ci se base sur une isolation des espaces adressables en modifiant les réseaux sur puces et non sur des extensions de virtualisation au sein des cœurs de calculs. Cela permet ainsi de garder la taille et la complexité des cœurs petite, ce qui est important pour une architecture manycore. De plus, la solution permet d'exécuter des systèmes d'exploitation invités directement sur les cœurs de la plateforme, sans couche de virtualisation, ce qui permet d'augmenter les performances des machines virtuelles [79, 46].

Pour réaliser le mécanisme d'isolation, les auteurs proposent une modification de la configuration des *Look-Up Tables* (LUTs) au sein de l'architecture SCC. Ces tables permettent de traduire les adresses physiques en adresses du système comme le montre la figure 3.9. Les LUTs se trouvent dans l'interface avec le réseau sur puce de chaque tuile de l'architecture. Deux types d'adresses système existent : les adresses internes à la tuile (accès à la mémoire locale, aux registres et aux LUTs de la tuile) et externes (accès aux périphériques, mémoire externes ou d'autres tuiles).

La figure 3.10 présente les différentes modifications apportées à l'architecture SCC. Le principal mécanisme de cette solution réside dans la modification des accès à l'espace de configuration des LUTs. En effet, dans l'architecture SEMA, tous les cœurs de la plateforme sont interdits d'accès à la configuration des LUTs. Un seul cœur, nommé *core manager*, est capable d'accéder à cette interface de configuration. Ce cœur exécute l'hyperviseur et est le seul capable de configurer les LUTs et donc de fournir l'isolation des différentes machines virtuelles. La granularité du déploiement des machines virtuelles est au niveau de la tuile, autrement dit deux machines virtuelles ne peuvent être sur la même tuile. L'autre modification apportée à l'architecture Intel SCC réside dans l'ajout d'une interface de démarrage permettant d'utiliser le processeur SCC comme un processeur *stand-alone* et non plus comme un co-processeur.

La solution proposée par R.J.Masti et al. est très proche de celle proposée dans cette thèse et a été développée en même temps. Néanmoins, nous pouvons noter que cette solution ne propose pas de mécanismes interdisant à l'hyperviseur d'accéder aux ressources allouées aux machines virtuelles.

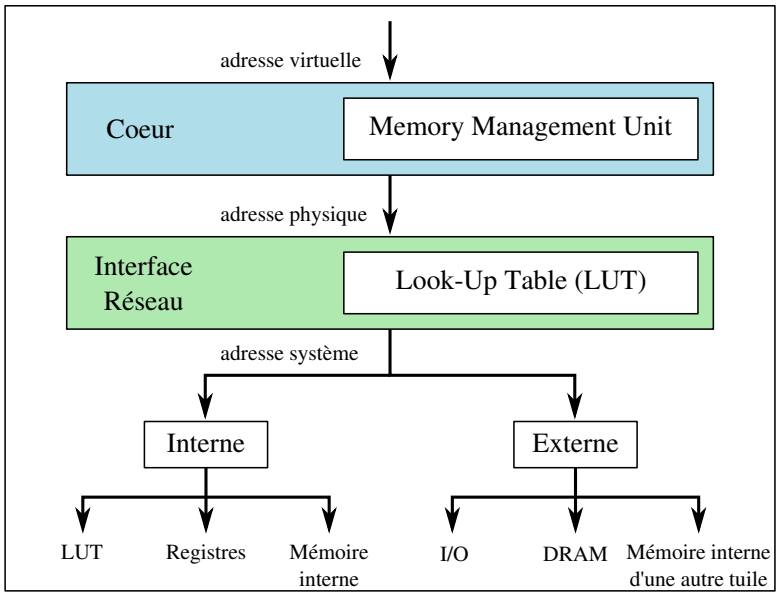

FIGURE 3.9 – Mécanisme de traduction d'adresse dans l'architecture Intel SCC
Source : R. J. Masti et al. [38]

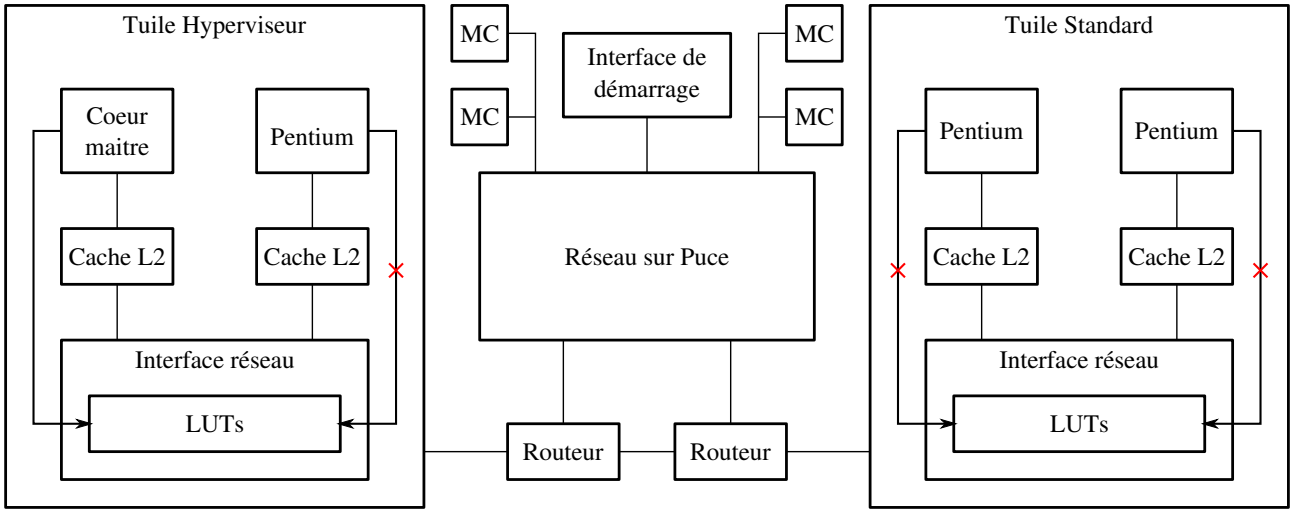

FIGURE 3.10 – Architecture SEMA
Source : R. J. Masti et al. [38]

De plus, l'architecture SEMA ne propose pas de cohérence matérielle, ce que nous jugeons difficilement réaliste dans le contexte d'exécution de systèmes d'exploitation multi-core génériques. D'ailleurs, dans leurs expérimentations, R. J. Masti et al. n'ont pas déployé de système d'exploitation multi-core mais seulement des systèmes mono-coeur.

3.3.7 Intel-SGX

V. Costan et S. Devadas expliquent dans leur article [80] le fonctionnement de la technologie *Software Guard Extensions* (SGX) d'Intel. Cette technologie est un ensemble d'extensions ajoutées à l'architecture Intel qui visent à garantir l'intégrité et la confidentialité des données d'une application sensible exécutée sur un serveur distant, sur lequel les autres entités logicielles privilégiées, tels que les systèmes d'exploitation ou l'hyperviseur, sont potentiellement malicieuses. Leurs analyses ont été réalisées à l'aide de trois articles [81, 82, 83], du *Intel Software Developer Manual* [84] et de deux brevets [85, 86].

La technologie Intel-SGX vise à résoudre le problème lié à la sécurité de l'utilisation d'un ordinateur distant en implantant des modules matériels de confiance dans les ordinateurs distants. Ces modules vont alors établir des conteneurs sécurisés et les utilisateurs vont charger le code de leurs applications ainsi que leurs données dans ces conteneurs. Les modules de confiance sont alors en charge d'assurer la confidentialité et l'intégrité de l'ensemble des données contenues dans les conteneurs. Intel-SGX se base sur l'authentification du logiciel, comme ses prédécesseurs TPM [87] et TXT [88]. L'authentification permet de prouver à l'utilisateur qu'il communique avec une entité logicielle s'exécutant dans la zone sécurisée hébergée par le matériel de confiance.

Intel-SGX déploie des zones de sécurité, ou enclaves, qui vont être protégées en isolant le code et les données du reste de la plateforme. Pour cela Intel-SGX réserve une partie de la mémoire, nommée *Processor Reserved Memory* (PRM), et le CPU interdit tous les accès à cette mémoire par des entités ne s'exécutant pas dans une enclave. Le PRM contient les *Enclave Page Cache* (EPC), qui sont un ensemble de pages de 4 Ko contenant le code et les données des logiciels s'exécutant dans les enclaves. Le logiciel système en charge d'attribuer les EPCs aux enclaves peut être soit l'hyperviseur, soit un système d'exploitation, et il n'est pas considéré comme étant de confiance. Ce logiciel peut aussi évincer les pages pour les stocker dans la mémoire non protégée. Le CPU met alors en place des protections cryptographiques pour assurer l'intégrité et la confidentialité de ces pages. Le CPU traque l'état de chaque page des EPCs dans les *Enclave Page Cache Metadata* (EPCM) pour assurer que chaque EPC appartient bien à une seule enclave, et donc vérifier que l'allocation des EPCs faite par le logiciel système est correcte, i.e. qu'une EPC n'appartient pas à deux enclaves différentes. La figure 3.11 illustre les différents éléments présentés ci-dessus.

Durant l'étape de chargement, le logiciel système demande au CPU de copier le code et les données de l'application à sécuriser, depuis la mémoire non protégée (hors PRM) dans les pages EPCs. Une fois que toutes les pages sont copiées, le CPU marque l'enclave comme initialisée, et à ce moment là l'application peut s'exécuter dans l'enclave. Pendant le chargement de l'enclave, son contenu est *hashé* pour permettre l'authentification de l'enclave. Ainsi, un utilisateur peut à n'importe quel moment vérifier qu'il communique bien avec l'enclave et que le logiciel s'exécute bien dans un environnement sécurisé.

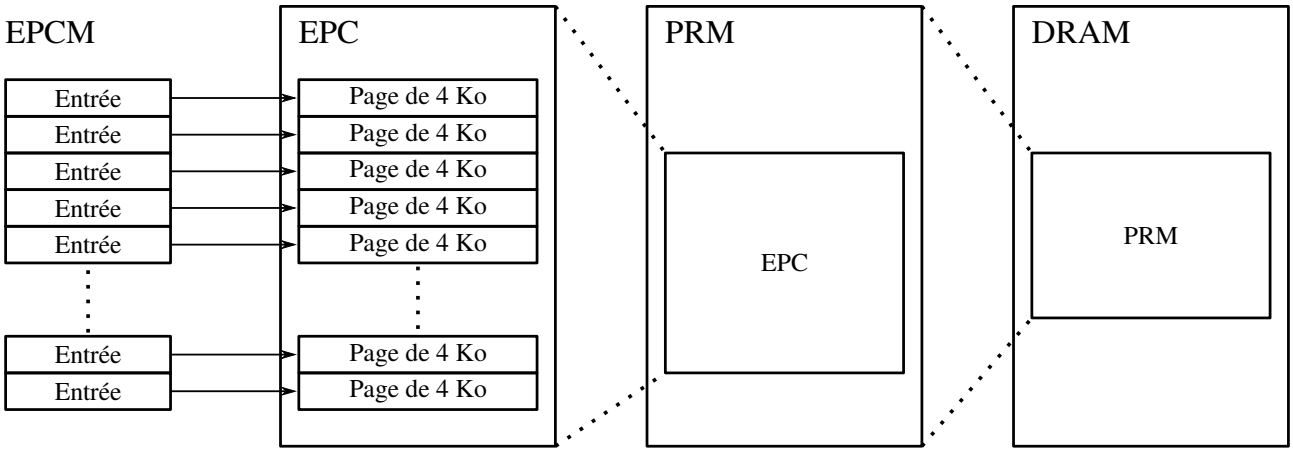

FIGURE 3.11 – Représentation des différents éléments PRM, EPC et EPCM
***Source** : V. Costan et S. Devadas [80]*

Pour exécuter le flot d'instructions d'une enclave, il faut exécuter des instructions spécifiques. Cela induit donc l'ajout d'instructions au sein du processeur. Le traitement de ces nouvelles instructions est réalisé à l'aide de microcodes, ce qui implique une modification des systèmes d'exploitation et des applications. Le développeur doit explicitement faire appel à l'exécution sécurisée au sein des enclaves.

La technologie Intel-SGX permet de créer des zones de sécurité dans un environnement considéré comme malicieux. L'hyperviseur, ou tout logiciel s'exécutant dans un niveau privilégié, ne peut pas accéder aux données des applications s'exécutant dans les enclaves. Par contre, cette technologie, par l'ajout d'instructions spécifiques, oblige les développeurs à modifier leurs applications et les systèmes d'exploitation. De plus, l'hyperviseur et les machines virtuelles s'exécutent sur les mêmes ressources de calculs, offrant ainsi une vulnérabilité aux attaques par canaux cachés. Enfin, la complexité des cœurs intégrant la technologie Intel-SGX permet difficilement une intégration sur des plateformes manycores.

3.4 Conclusion

Au cours de ce chapitre nous avons présenté six architectures sécurisées qui cherchent à résoudre le problème d'exécution de machines virtuelles de manière sécurisée. La table 3.1 résume les différentes caractéristiques de ces architectures, au nombre de cinq :

- Est-ce que la solution apportée ajoute de la complexité au cœur ?
- Est-ce que la solution nécessite une modification des logiciels existants, à savoir les systèmes d'exploitation invités ou l'hyperviseur ?
- Est-ce que la solution nécessite un partage des ressources entre les machines virtuelles et l'hyperviseur ?

– Quelle est la TCB de la solution ?
– L'hyperviseur a-t-il des droits limités ?

Nom	Complexification des cœurs	Modifications logicielles	Partage de ressources	TCB	Limitation hyperviseur
Bastion	Oui	Apps. + OS	Oui	CPU + Hyp.	Non
NoHype	Oui	OS	Oui	CPU + Firmware	Pas d'hyp.
HyperWall	Oui	Hyp.	Oui	CPU	Oui
H-SVM	Oui	Hyp. + OS	Oui	CPU	Oui
SEMA	Non	Aucune	Non	CPU + Hyp.	Non
Intel-SGX	Oui	Apps. + OS	Oui	CPU	Oui

TABLE 3.1 – Résumé des différentes architectures sécurisées présentées

Nous remarquons qu'aucune des solutions présentées ne répond à nos exigences, à savoir : une complexité des cœurs faible, des piles logicielles non modifiées, une isolation stricte des ressources entre les machines virtuelles et l'hyperviseur, une TCB excluant l'hyperviseur, et des droits d'accès limités pour l'hyperviseur.

Mes travaux cherchent à unifier ces approches en prenant le problème dans sa globalité sur un manycore cc-NUMA réaliste utilisant des périphériques sur un bus dédié. Plus précisément, le but du projet est de pouvoir lancer et arrêter de manière dynamique des machines virtuelles sur l'architecture manycore TSAR en les isolant par une MMU segmentée, nommée *Hardware Address Translator* (HAT). Cette MMU segmentée effectue la traduction entre deux espaces d'adressage physique (hôte et invité), et permet l'isolation de la mémoire et des canaux de périphériques. Celle-ci gère aussi la cohérence entre les caches L1 sachant que les L1 et les L2 ne sont plus dans le même espace d'adressage. Les ressources (processeurs, mémoires et canaux de périphériques) sont allouées aux machines virtuelles par l'hyperviseur de manière exclusive comme dans la thèse de S. Leroy, ou dans les travaux de R. J. Masti, et sous le contrôle des MMU segmentées. Un point important est que l'hyperviseur est spécifié de manière à avoir le minimum de droits de manière à ne pas pouvoir accéder aux données d'une machine virtuelle une fois que celle-ci est démarrée.

Chapitre

4 *Solutions*

Contents

Dans ce chapitre nous allons présenter les différents mécanismes matériels et logiciels mis en place au cours de cette thèse permettant de répondre aux problèmes relevés dans le chapitre 2. Nous présenterons dans la section 4.1 l'architecture développée au cours de cette thèse, appelée architecture Tsunamy. Cette architecture se base sur l'architecture TSAR et a été réalisée avec des modèles SystemC au niveau *Cycle Accurate Bit Accurate* (CABA). Nous étudierons ensuite dans la section 4.2 notre solution matérielle permettant l'isolation physique des machines virtuelles ainsi que la virtualisation complète des systèmes d'exploitation invités. Notre solution consiste à ajouter un composant matériel, nommé *Hardware Address Translator*, devant chaque composant initiateur de l'architecture. Dans la section 4.3 nous traiterons des problèmes liés au partage des périphériques en utilisant la technique d'assignation directe. Ensuite, nous présenterons dans la section 4.4, notre mécanisme permettant de démarrer une machine virtuelle. Ce mécanisme se base sur une procédure logicielle ayant pour but d'assurer la bonne configuration du matériel en charge de l'isolation des machines virtuelles. Puis nous présenterons, dans la section 4.5, notre mécanisme d'arrêt des machines virtuelles. Ce mécanisme est une procédure logicielle assistée par le matériel, qui garantit qu'à la fin de son exécution, l'espace précédemment alloué à la machine virtuelle est libéré et réutilisable pour une autre allocation. En dernier, nous présenterons dans la section 4.6 notre solution de chiffrement de disque pour les machines virtuelles. Le chiffrement et le déchiffrement du disque sont assistés matériellement par un crypto-processeur, HCrypt, présent dans chaque cluster de l'architecture Tsunamy, permettant de grandement accélérer le processus en comparaison à une solution logicielle.

4.1 Architecture Tsunamy

La figure 4.1 présente l'architecture Tsunamy. Celle-ci est basée sur l'architecture TSAR développée au LIP6. Les différences matérielles entre les deux architectures sont le rajout des composants *Hardware Address Translator* (HAT), *Shutdown Virtual Machine Controller* (SVM Controller), *Shutdown Virtual Machine Agents* (SVM Agent) et des coprocesseurs cryptographiques (HCrypt). Les composants *Hardware Address Translator* seront détaillés dans la section 4.2 et les composants *Shutdown Virtual Machine Controller* et *Shutdown Virtual Machine Agents* dans la section 4.5.

Tsunamy est une architecture clusterisée avec une topologie *2D-Mesh* et utilisant un réseau sur puce (NoC). Le cluster ayant pour coordonnées $(0, 0)$ possède un accès vers les périphériques externes, et ce cluster est appelé *Cluster Hyperviseur*. Ce cluster est en effet dédié à l'hyperviseur, qui sera exécuté de manière exclusive sur ce cluster. Cette architecture, comme l'architecture TSAR, a été développée dans le but de pouvoir supporter jusqu'à 256 (16×16) clusters.

Chaque cluster de l'architecture contient :

– 4 cœurs MIPS-32 avec leur *Memory Management Unit* (MMU) et leur premier niveau de cache (L1), séparé entre instructions et données. La cohérence du cache L1 est entièrement gérée en matériel. Les *miss* dans le *Translation Lookaside Buffer* (TLB) sont aussi gérés en matériel.

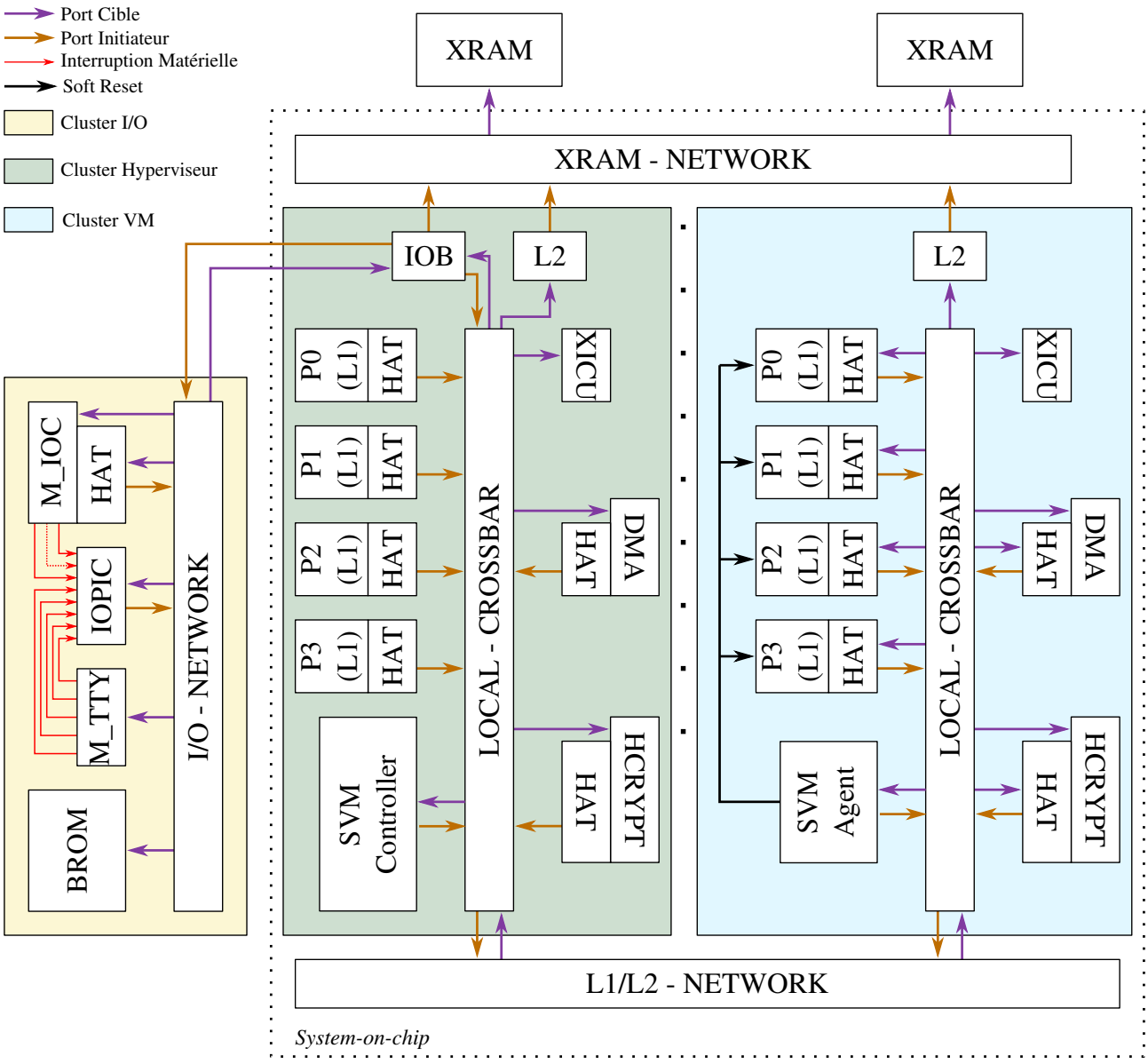

FIGURE 4.1 – Schéma de l'architecture Tsunamy

- Un cache de second niveau (L2), qui est responsable d'un segment de l'espace mémoire de la machine. En particulier, le cache L2 est responsable de la cohérence des copies, dans les différents caches L1, des lignes contenues dans ce segment.
- 3 périphériques internes : un contrôleur d'interruptions incluant des fonctions de *timer* (XICU), un contrôleur *Direct Memory Access* (DMA) ainsi qu'un coprocesseur cryptographique (HCrypt). Ces périphériques sont nommés périphériques répliqués.
- Des *Hardware Address Translators* devant chaque composant initiateur sur le réseau. Ces composants permettent l'isolation des machines virtuelles et de l'hyperviseur. Ceux-ci sont détaillés dans la section 4.2.
- Un *crossbar* local permettant d'interconnecter ces composants avec un accès vers le réseau global (réseau L1/L2) par le biais d'un routeur.
- Un composant SVM : *Controller* pour le cluster hyperviseur, et *Agent* pour les autres clusters.

Le cluster hyperviseur contient aussi un composant supplémentaire *Input/Output Bridge* (IOB) permettant l'accès au cluster contenant les périphériques, nommé Cluster I/O.

Le cluster I/O possède les composants suivants :
- Un contrôleur de terminaux multi-canaux (M_TTY).
- Un contrôleur de disque multi-canaux (M_IOC), ainsi que son *Hardware Address Translator* permettant de garantir l'intégrité des machines virtuelles étant donné que le M_IOC est initiateur sur le réseau.
- Un contrôleur d'interruption programmable (IOPIC) qui est capable de convertir une interruption matérielle en interruption logicielle. Ce composant est principalement utilisé pour pouvoir router les interruptions des canaux des périphériques vers les XICUs des machines virtuelles.
- Une *Read-Only Memory* de *boot* (BROM) contenant le code de l'hyperviseur et le code initialisant les cœurs à leur démarrage.
- Un réseau I/O permettant d'interconnecter tous les composants du cluster possédant un accès vers le réseau des RAMs (réseau XRAM) et le réseau global (réseau L1/L2) par le biais de l'IOB.

Cette architecture a été modélisée en SystemC au niveau CABA. Ainsi, toutes les évaluations concernant le coût matériel de notre solution ne sera qu'une estimation en termes de bits mémorisants, étant donné qu'aucune implémentation RTL n'a été réalisée.

4.2 Isolation des machines virtuelles

La technique employée pour isoler physiquement les machines virtuelles s'exécutant sur l'ensemble de l'architecture est l'ajout d'un nouvel espace d'adressage, où les adresses sont nommées adresses machine (figure 4.2). Les adresses machine émises par la MMU d'un cluster sont traduites en adresses physiques quand elles arrivent sur le *crossbar local* (réseau L1/L2). Le mécanisme de

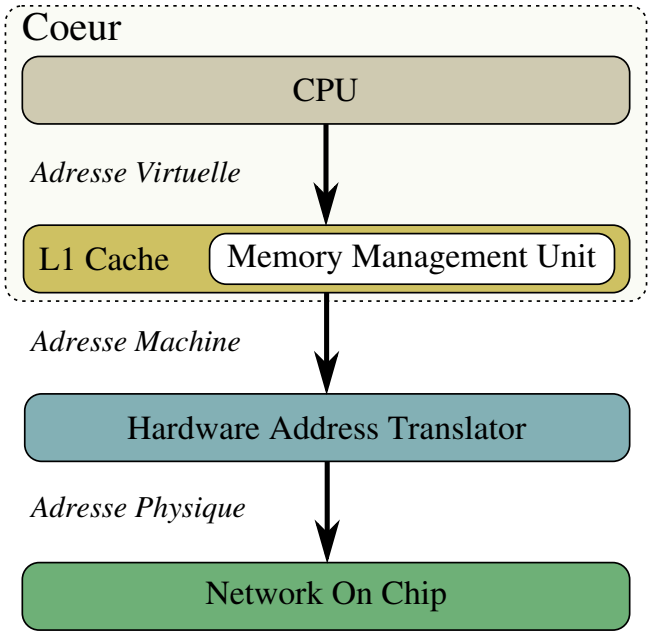

FIGURE 4.2 – Représentation des trois espaces d'adressage : Virtuel, Machine et Physique

traduction assure que toutes les adresses physiques obtenues pour une machine virtuelle pourront cibler seulement des ressources allouées à la machine virtuelle. Cette traduction est faite par le biais du composant *Hardware Address Translator*. Celui-ci est configuré par l'hyperviseur au démarrage de la machine virtuelle et est placé devant chaque initiateur de l'architecture – aussi bien les cœurs que les périphériques répliqués comme les DMAs.

Le composant *Hardware Address Translator* joue le même rôle qu'une MMU segmentée mais diffère sur certains points, en particulier le fait qu'il utilise des informations de topologies pour réaliser la traduction d'adresse. L'inconvénient de ce choix comparé à la pagination est une perte de flexibilité dans l'allocation des ressources, une machine virtuelle ne peut être déployée que sur un cluster au minimum et les clusters alloués doivent être contigus et convexes. D'un point de vue sécurité, cela permet d'isoler physiquement les clusters des différentes machines virtuelles par le biais du routage d'adresses (à l'exception des périphériques). En résumé, notre solution pour l'isolation physique est un module de segmentation simple et rapide, dont le principal défaut est la granularité de traduction.

Le composant *Hardware Address Translator* possède deux états qui alternent au cours de son utilisation. La figure 4.3 montre cette alternance entre les deux états. Lorsque le processeur démarre, les *Hardware Address Translators* sont dans l'état *désactivé* dans lequel ils fournissent une traduction identique entre les adresses machine et physiques. Dans cet état, ils sont configurables par n'importe quel cœur ; en pratique, cela ne peut être qu'un cœur inactif, i.e. un cœur non dédié à l'hyperviseur ou exécutant une machine virtuelle. Lors de la séquence de démarrage d'une machine virtuelle, ils sont configurés par un des cœurs de la machine virtuelle. Puis, ils sont activés par leur propre cœur, et ils changent leur état en *activé*. Une fois dans l'état *activé* les *Hardware Address Translators* ne peuvent

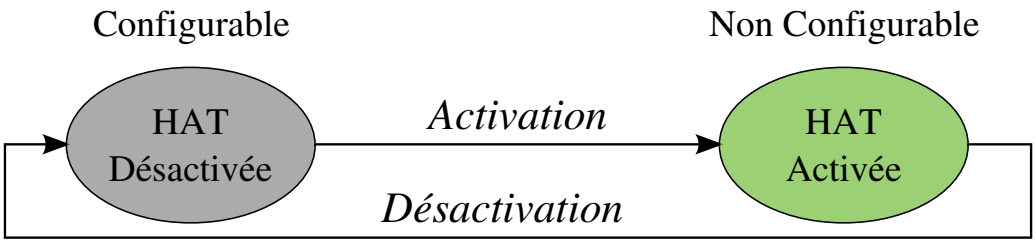

FIGURE 4.3 – Les différents états d'un *Hardware Address Translator*

plus être reconfigurés ; une nouvelle configuration est seulement possible après leur désactivation, ce qui est indissociable de l'arrêt de la machine virtuelle.

Comme le montre la figure 4.4, le *Hardware Address Translator* possède deux mécanismes de fonctionnement en fonction du type de l'adresse machine émise par l'initiateur :

- Soit l'adresse cible un périphérique contenu dans un des clusters alloués à la machine virtuelle : mémoire *via* le cache L2 ou un périphérique répliqué tel qu'un DMA, une XICU. C'est le cas le plus courant et nous l'appelons *accès interne*. Ce mécanisme est expliqué en détail dans la sous-section 4.2.2.
- Soit l'adresse cible un périphérique contenu à l'extérieur des clusters alloués à la machine virtuelle, tel que le contrôleur de disque, le TTY ou le réseau Ethernet ; ce type d'accès est appelé *accès externe*. Ce mécanisme est expliqué en détail dans la sous-section 4.2.3.

La figure 4.5 donne la vue interne du composant *Hardware Address Translator*. Ce composant matériel configurable est placé devant tous les initiateurs, c'est-à-dire les composants ayant la capacité d'initier un transfert (couple requête/réponse). Pour pouvoir s'interfacer entre les initiateurs et le *crossbar* local, il possède un port VCI cible du côté de l'initiateur devant lequel il est placé, et un port VCI initiateur du côté du *crossbar* local. Il possède aussi un port VCI cible permettant sa configuration. Un *Hardware Address Translator* possède trois couples de ports DSPIN cible-initiateur pour permettre la traversée des requêtes de cohérence du protocole DHCCP utilisé dans l'architecture. En effet, ce composant doit traduire aussi bien les requêtes directes VCI que les requêtes de cohérence

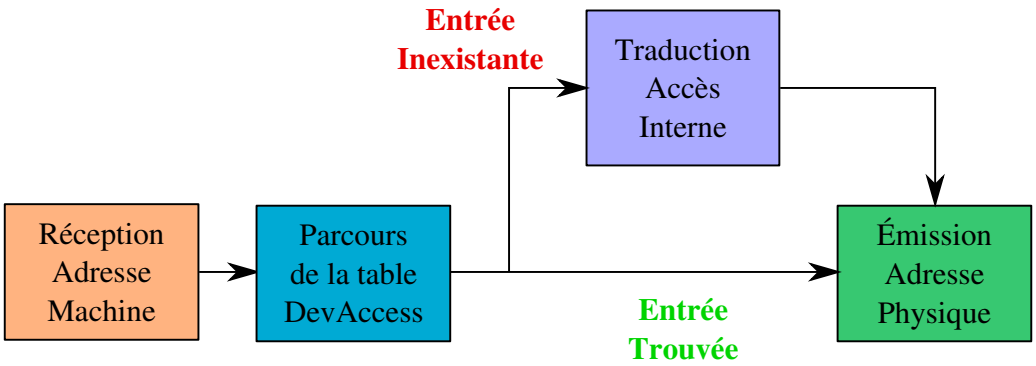

FIGURE 4.4 – Les différents modes de fonctionnement d'un *Hardware Address Translator*

car les lignes de cache stockées dans un cache L1 sont dans l'espace machine alors que les lignes correspondantes dans les caches L2 sont stockées dans l'espace physique de la plateforme. Comme le montre la figure 4.5, nous pouvons compartimenter un *Hardware Address Translator* en six parties.

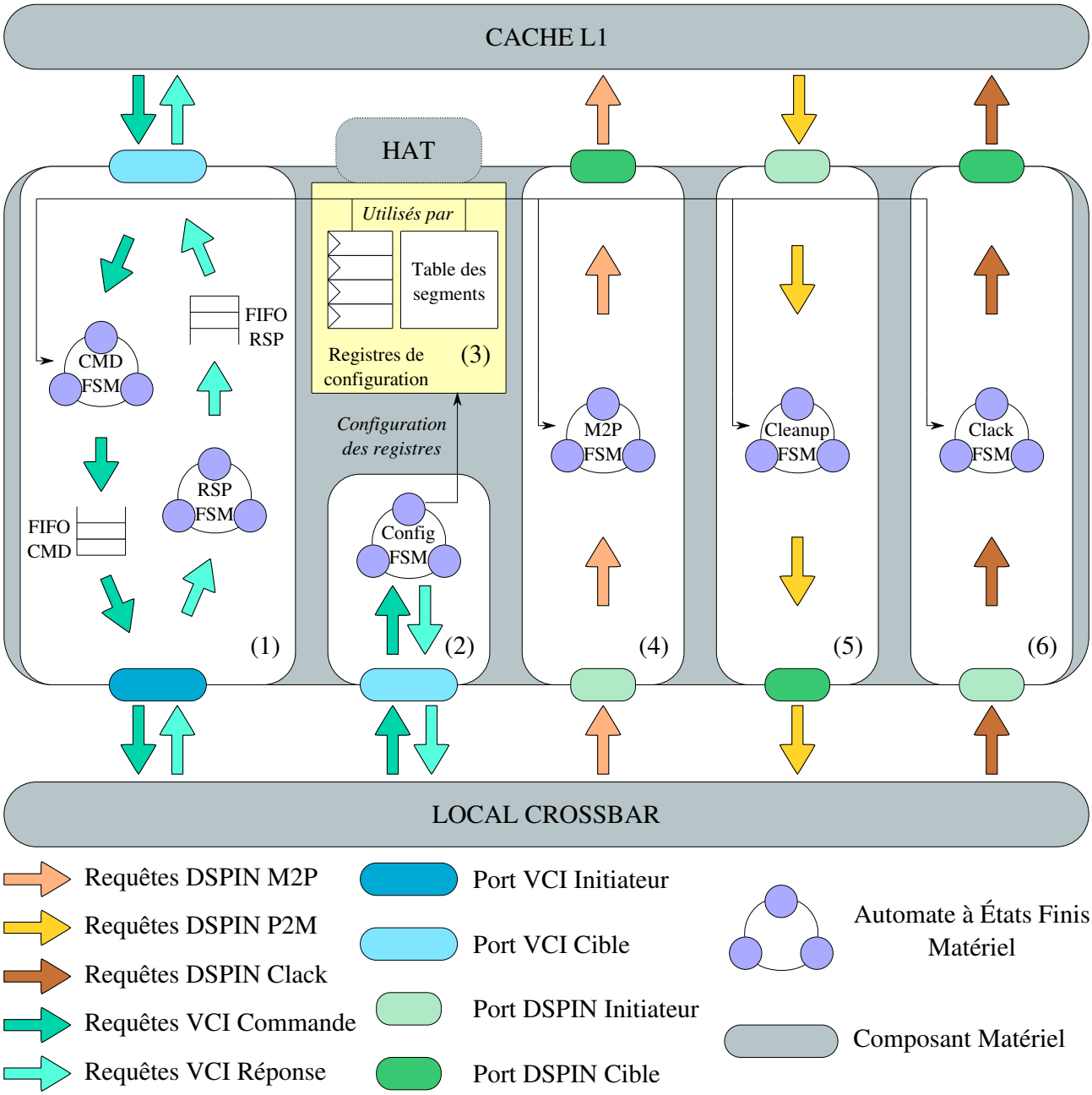

FIGURE 4.5 – Vue globale d'un *Hardware Address Translator*

Le compartiment (1) est en charge de traduire les requêtes directes provenant des initiateurs. Celui-ci contient principalement deux automates d'états finis, `CMD_FSM` et `RSP_FSM` ainsi que deux FIFOs. L'automate `CMD_FSM` est en charge de recevoir les requêtes provenant de l'initiateur et de les traduire en utilisant les mécanismes d'accès internes ou externes. L'automate `RSP_FSM` est lui en charge de la réception des réponses des requêtes précédemment envoyées par l'automate `CMD_FSM` et donc

par l'initiateur lui-même.

Le compartiment (2) contient l'automate CONFIG_FSM qui est en charge de la configuration d'un *Hardware Address Translator*. En fonction des requêtes reçues sur le port VCI cible CONFIG, l'automate CONFIG_FSM va venir stocker les différentes configurations dans les registres correspondants et les tables de segments, contenus dans le compartiment (3) du composant. Les différents registres de configuration seront détaillés dans la sous-section 4.2.1.

Le compartiment (4) contient l'automate M2P_FSM qui est en charge de recevoir les requêtes de cohérence provenant des caches L2 en direction des caches L1. Cet automate doit traduire les numéros de ligne physique en numéro de ligne machine, permettant ainsi le bon fonctionnement du protocole de cohérence. Les messages de cohérence traités par cet automate sont : MULTI_INVAL, BROADCAST_INVAL et MULTI_UPDATE. Le compartiment (5) contient l'automate CLEANUP_FSM qui est en charge des requêtes de cohérence provenant d'un cache L1 vers un cache L2. Cet automate doit donc traduire des numéros de lignes machine en numéros de ligne physique. Les messages de cohérence traités par cet automate sont : CLEANUP et MULTI_ACK. Enfin le compartiment (6) contient l'automate CLACK_FSM en charge de recevoir les messages CLACK émis par un cache L2 pour un cache L1.

Le coût matériel d'un composant *Hardware Address Translator* est de 182 octets de mémoire : 5 octets pour les registres de configuration, 10 octets pour chacune des 5 entrées de la table des segments, 85 octets pour les registres internes utilisés par les différents automates, et 42 octets pour les deux FIFOs VCI de profondeur 2. En comparaison, la taille d'une ligne de cache de l'architecture est de 64 octets.

La figure 4.6 permet d'illustrer le problème lié à la cohérence dans le cas où un cache L1 n'est pas dans le même espace d'adressage qu'un cache L2. Soient deux lignes de cache stockées en adresse machine au sein du cache L2, 0xC0BD0000 et 0xC0BD0040, et leurs copies stockées en adresse machine dans deux caches L1, 0x40BD0000 et 0x40BD0040. Si le cache L2 cherche à invalider la ligne 0xC0BD0040, celui-ci va alors envoyer un message MULTI_INVAL contenant le numéro de la ligne 0xC0BD0040. Lorsque le cache L1 du cœur *C1* reçoit ce message, il va alors vérifier si l'invalidation reçue correspond à une des lignes contenue dans son répertoire. Si le *Hardware Address Translator* n'a pas réalisé la traduction inverse, machine vers physique, la comparaison entre 0xC0BD0040 et sa correspondance 0x40BD0040 va échouer et la ligne ne sera pas évincée du cache L1. Cela provoquera par la suite un blocage de la machine, car le cache L2 s'attendait à ce que la ligne 0xC0BD0040 existe dans le cache L1 du cœur *C1*.

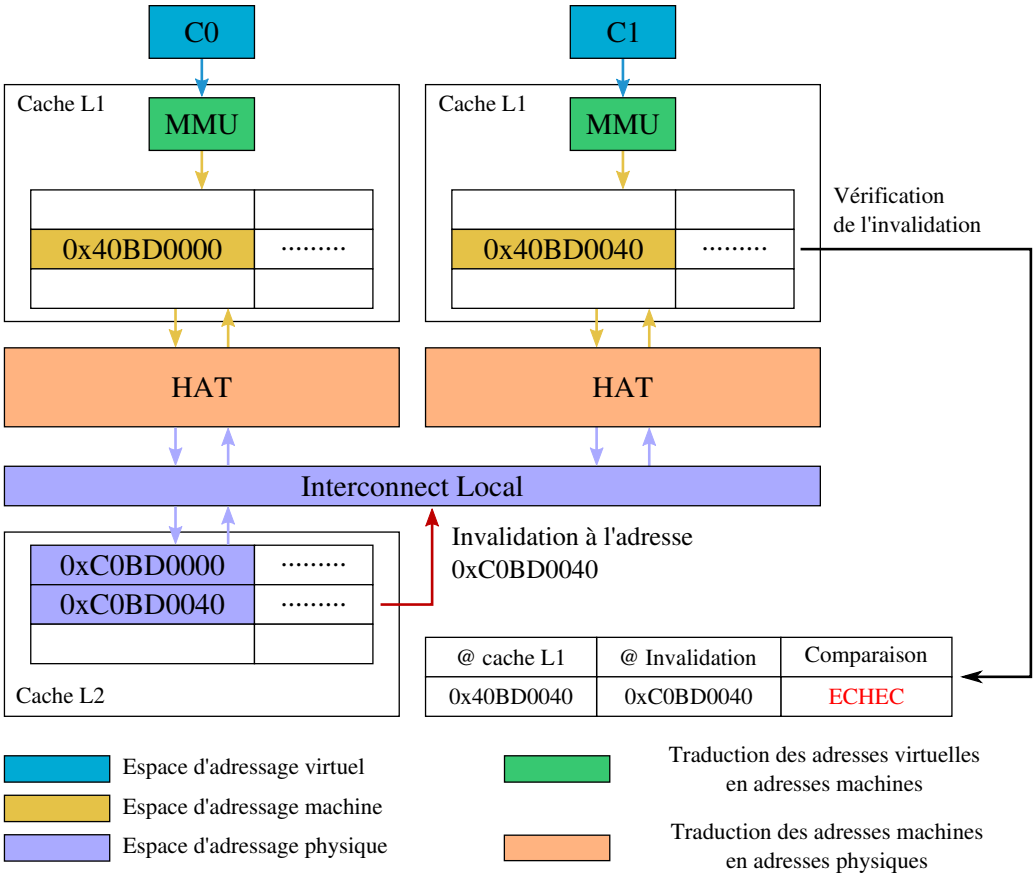

FIGURE 4.6 – Illustration du problème de la cohérence posé par l'existence de deux domaines d'adressage différents pour les caches L1 et L2

4.2.1 Registres de configuration

Comme dit précédemment, les *Hardware Address Translators* sont des composants configurés au démarrage d'une machine virtuelle par l'hyperviseur. Pendant la séquence d'initialisation d'une machine virtuelle, l'hyperviseur doit donc configurer un certain nombre de registres des *Hardware Address Translators*. Le tableau 4.1 présente ces différents registres configurables.

`HAT_MODE` : ce registre sert à définir le mode de fonctionnement d'un composant. Il existe trois valeurs possibles pour ce registre : *IDENTITY*, *ACTIVATE_32* et *ACTIVATE_40*. Le mode *IDENTITY* est le mode dans lequel l'adresse émise en sortie est celle donnée en entrée, autrement dit le mode dans lequel le *Hardware Address Translator* est désactivé. Les modes *ACTIVATE_32* et *ACTIVATE_40* sont respectivement les modes dans lesquels le composant fournit une traduction des adresses machine vers les adresses physiques pour des systèmes d'exploitation invités 32 bits et 40 bits. Lorsqu'un *Hardware Address Translator* est désactivé, i.e. dans le mode *IDENTITY*, celui-ci est configurable. Lorsque l'on vient l'activer via les modes *ACTIVATE_32* ou *ACTIVATE_40*, un registre interne `R_CONFIG_LOCK` est mis à 1 et empêche la reconfiguration du composant. Seule la procédure d'arrêt d'une machine virtuelle peut alors venir désactiver ce composant *Hardware Address Translator* actif.

Nom du registre	Permission d'accès	Taille (en bits)
HAT_MODE	Écriture	2
HAT_PX0	Écriture	4
HAT_PY0	Écriture	4
HAT_MXL	Écriture	4
HAT_MYL	Écriture	4
HAT_PBA	Écriture	32 par entrée
...	...	
HAT_MASK	Écriture	32 par entrée
...	...	
HAT_PBA_EXT	Écriture	8 par entrée
...	...	
HAT_MASK_EXT	Écriture	8 par entrée
...	...	

TABLE 4.1 – Table des registres configurables des *Hardware Address Translators*

HAT_PX0 : ce registre sert à stocker l'information de la coordonnée physique X du cluster $(0,0)$ de la machine virtuelle. Cette information est utilisée par l'automate CMD_FSM, M2P_FSM, CLEANUP_FSM et CLACK_FSM pour le mécanisme de traduction des accès internes et des requêtes de cohérence.

HAT_PY0 : ce registre sert à stocker l'information de la coordonnée physique Y du cluster $(0,0)$ de la machine virtuelle. Cette information est utilisée par l'automate CMD_FSM, M2P_FSM, CLEANUP_FSM et CLACK_FSM pour le mécanisme de traduction des accès internes et des requêtes de cohérence.

HAT_MXL : ce registre sert à stocker l'information du nombre de bits nécessaire pour coder la taille en coordonnée X de la machine virtuelle. Cette information est utilisée par l'automate CMD_FSM, M2P_FSM, CLEANUP_FSM et CLACK_FSM pour le mécanisme de traduction des accès internes et des requêtes de cohérence.

HAT_MYL : ce registre sert à stocker l'information du nombre de bits nécessaire pour coder la taille en coordonnée Y de la machine virtuelle. Cette information est utilisée par l'automate CMD_FSM, M2P_FSM, CLEANUP_FSM et CLACK_FSM pour le mécanisme de traduction des accès internes et des requêtes de cohérence.

HAT_PBA : ce banc de registre sert à stocker les 32 bits de poids faible des adresses physiques de base des périphériques alloués aux machines virtuelles. Ce banc de registre possède une taille dépendant du nombre de périphériques maximum que contient la plateforme. Il est utilisé par l'automate CMD_FSM lors d'un accès externe par une machine virtuelle.

`HAT_MASK` : ce banc de registre sert à stocker les 32 bits de poids faibles du complément à deux de la taille des segments des périphériques alloués aux machines virtuelles. Ce banc de registre possède une taille dépendant du nombre de périphériques maximum que contient la plateforme. Il est utilisé par l'automate `CMD_FSM` lors d'un accès externe par une machine virtuelle.

`HAT_PBA_EXT` : ce banc de registre sert à stocker les 8 bits de poids forts, dits bits d'extension, des adresses physiques de base des périphériques alloués aux machines virtuelles. Ce banc de registre possède une taille dépendant du nombre de périphériques maximum que contient la plateforme. Il est utilisé par l'automate `CMD_FSM` lors d'un accès externe par une machine virtuelle.

`HAT_MASK_EXT` : ce banc de registre sert à stocker les 8 bits de poids forts des compléments à deux de la taille des segments des périphériques alloués aux machines virtuelles. Ce banc de registre possède une taille dépendant du nombre de périphériques maximum que contient la plateforme. Il est utilisé par l'automate `CMD_FSM` lors d'un accès externe par une machine virtuelle.

4.2.2 Accès internes

L'espace d'adressage machine est distribué sur les clusters d'une telle manière que les bits de poids fort de l'adresse machine (MSB-machine) définissent les coordonnées du cluster machine dans la machine virtuelle. C'est exactement comme dans l'architecture TSAR, sauf que dans le cas de TSAR, il s'agit des adresses physiques. Plus précisément, les adresses machine générées par la MMU sont sur 40 bits mais les 8 bits d'extension (bits 39 à 32) sont toujours à 0 car le système d'exploitation est un système 32 bits. Le MSB-machine est placé avant le bit 32 et le nombre de bits du MSB-machine dépend du nombre de clusters de la machine virtuelle. De ce fait, la traduction des adresses machine en adresses physiques consiste à calculer les coordonnées du cluster physique de destination de la machine virtuelle dans l'espace physique à partir des bits de poids fort de l'adresse 32 bits (MSB machine) et à écrire les coordonnées de ce cluster physique dans les 8 bits de poids fort de l'adresse physique sur 40 bits.

La figure 4.7 illustre le principe général du mécanisme des accès internes pour une plateforme à 16 clusters. Comme on peut le voir, les bits de poids forts de l'adresse machine, correspondant aux coordonnées machines du cluster ciblé (*MX* et *MY*), vont être traduits à l'aide d'une fonction de transposition $f(mx, my)$ pour ainsi devenir les coordonnées physiques du cluster ciblé par la requête, soit *PX* et *PY*.

Par exemple, si l'on considère la machine virtuelle en vert sur la figure 4.7 et déployée sur 4 clusters, nous remarquons que son cluster $(0, 0)$ en coordonnée machine est en réalité sur le cluster $(0, 2)$ de la plateforme. Ainsi, si la machine virtuelle veut émettre une requête en direction de son cluster $(0, 0)$, celle-ci va émettre une adresse machine avec les champs *MX* et *MY* à 0, mais le *Hardware*

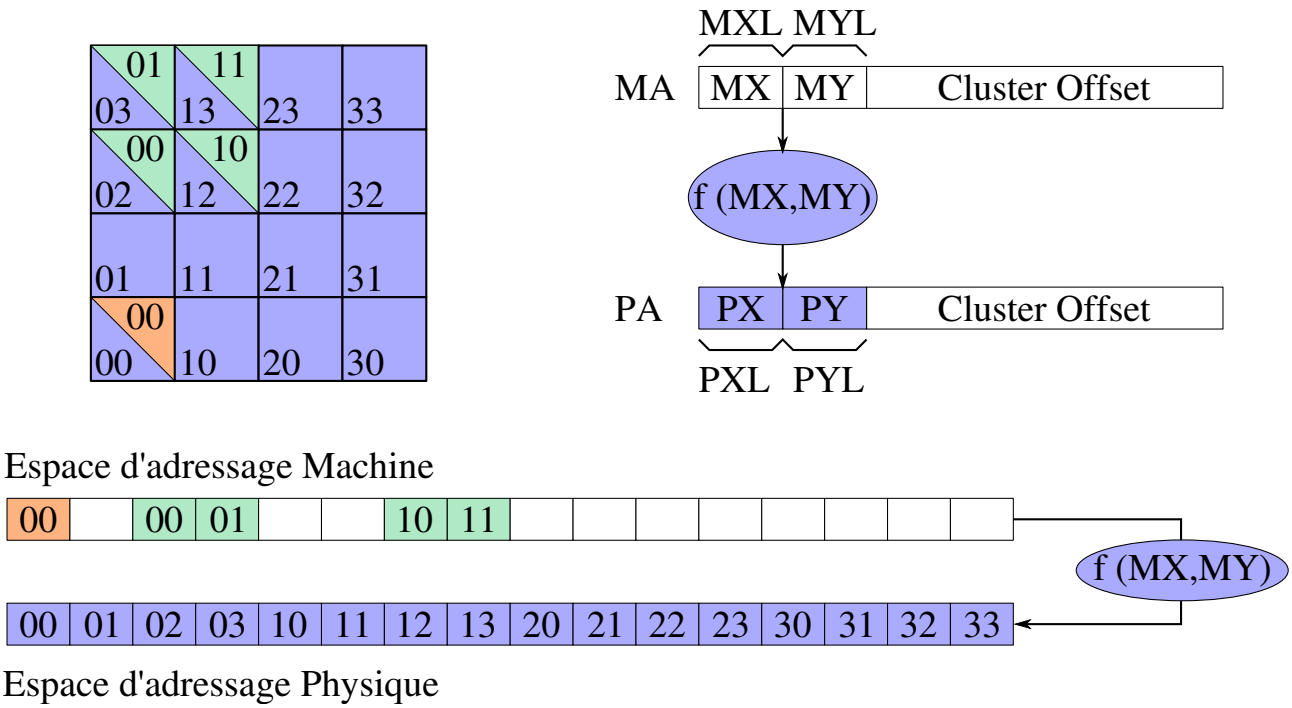

FIGURE 4.7 – Traduction d'une adresse machine en adresse physique pour un accès interne

Address Translator va venir modifier ces bits de poids fort pour les faire correspondre aux coordonnées physiques du cluster ciblé, soit ici $(2, 0)$. Ainsi, PX aura la valeur 0 et PY aura la valeur 2.

La fonction de traduction $f(mx, my)$ a pour expression :
- $f(mx, my) = (((px << pxl)|py) << (paddr_{width} - pxl - pyl))$
- $px = mx + px_0$
- $py = my + py_0$

Nous allons maintenant étudier deux cas de figure : un premier cas concernant la traduction effectuée dans le cas d'un système d'exploitation invité 32 bits puis 40 bits.

Exemple de traduction pour un système d'exploitation invité 32 bits

La figure 4.8 illustre le fonctionnement du module *Hardware Address Translator* sur une plateforme à 16 clusters pour un système d'exploitation invité 32 bits.

L'adresse est émise par un cœur dans une machine virtuelle s'exécutant sur les clusters 2, 3, 6 et 7 de la plateforme. La taille en X et en Y de la machine virtuelle est de 2. Le champ *MX* représente le nombre de bits nécessaires pour l'encodage de la taille en X de la machine virtuelle. Le champ *MY* est identique au champ *MX* mais pour la taille en Y. Dans notre exemple, les champs *MX* et *MY* valent 0 et 1 respectivement.

Le cœur envoie l'adresse virtuelle 0x83681424, qui est traduite par la MMU en l'adresse machine 0x41487424. Dans une machine virtuelle à 4 clusters, l'adresse machine commençant par 0x4 est située dans son deuxième cluster. En effet si on décompose 0x4 en binaire nous avons 0b0100, ce qui veut dire que le champ *MX* vaut 0 et le champ *MY* vaut 1 : l'adresse machine a donc pour destination le cluster $(0,1)$ de la machine virtuelle. Cela correspond effectivement à son deuxième cluster, soit le cluster $(0,3)$ de la plateforme. Le *Hardware Address Translator* connait le cluster 0 de la machine virtuelle, ici le cluster $(0,2)$ de la plateforme, ainsi que la topologie de son déploiement, ici 2 clusters en X et 2 clusters en Y. Il remplace alors les bits de poids fort *PX* et *PY* sur la figure 4.8 par les véritables coordonnées physiques du cluster cible de la requête, soit ici le cluster $(0,3)$.

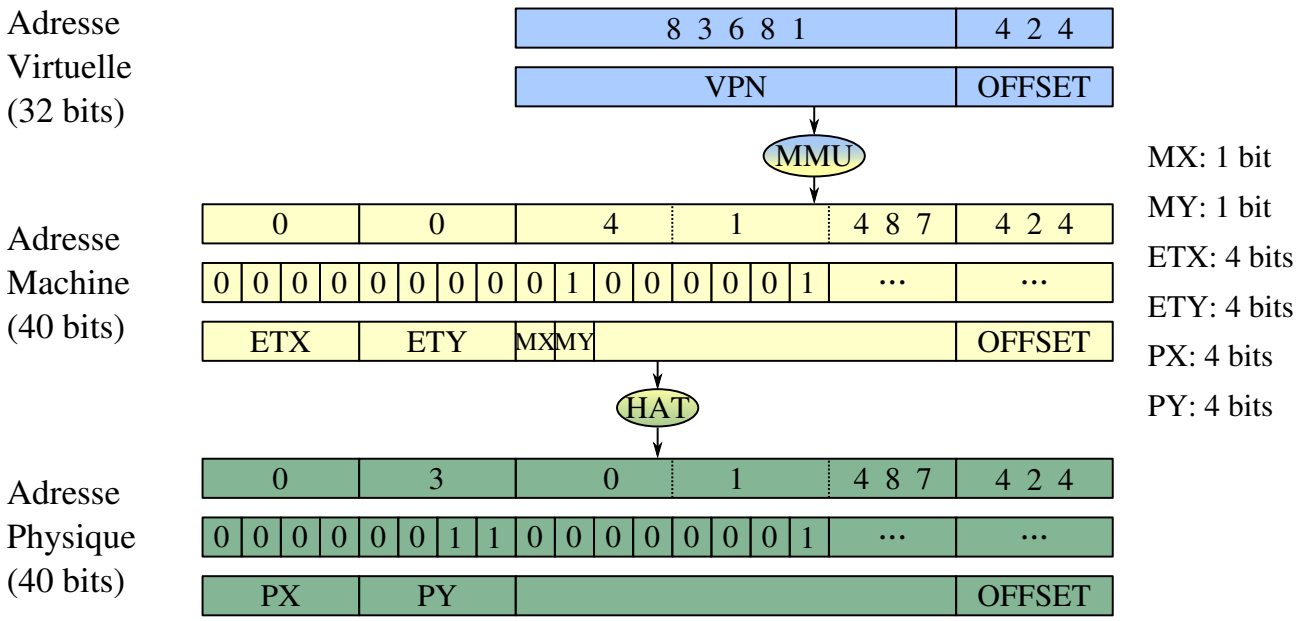

FIGURE 4.8 – Traduction d'une adresse machine en adresse physique pour un accès interne avec un système d'exploitation invité 32 bits

Exemple de traduction pour un système d'exploitation invité 40 bits

La figure 4.9 illustre le fonctionnement du module *Hardware Address Translator* sur une plateforme à 16 clusters pour un système d'exploitation invité 40 bits.

L'adresse est émise par un cœur dans une machine virtuelle s'exécutant sur les clusters 2, 3, 6 et 7 de la plateforme. La taille en X et en Y de la machine virtuelle est de 2. Pour les systèmes d'exploitations fonctionnant en 40 bits, les champs *MX* et *MY* sont de taille fixe et sont égaux aux tailles des bits d'extension *ETX* et *ETY*. Ainsi, les champs *MX* et *MY* ont la valeur 4. Le mécanisme de traduction est alors simplifié – par rapport à celui pour les systèmes d'exploitations 32 bits – car il suffit de venir modifier la valeur de *MX* et *MY* par les coordonnées physiques *PX* et *PY*.

Le cœur envoie l'adresse virtuelle 0x83681424 qui est traduite par la MMU en l'adresse machine 0x0101487424. Dans une machine virtuelle à 4 clusters, l'adresse machine commençant par 0x01 est située dans son deuxième cluster. En effet si on décompose 0x01 en binaire cela donne 0b00000001, ce qui veut dire que le champ *MX* vaut 0 et que le champ *MY* vaut 1 : l'adresse machine a donc pour destination le cluster $(0,1)$ de la machine virtuelle. Cela correspond effectivement à son deuxième cluster, soit le cluster $(0,3)$ de la plateforme. Le *Hardware Address Translator* connait le cluster 0 de la machine virtuelle, ici le cluster $(0,2)$ de la plateforme, ainsi que la topologie de son déploiement, ici 2 clusters en X et 2 clusters en Y. Il remplace alors les bits de poids fort *MX* et *MY* sur la figure 4.9 par les véritables coordonnées physiques du cluster cible de la requête, soit ici le cluster $(0,3)$.

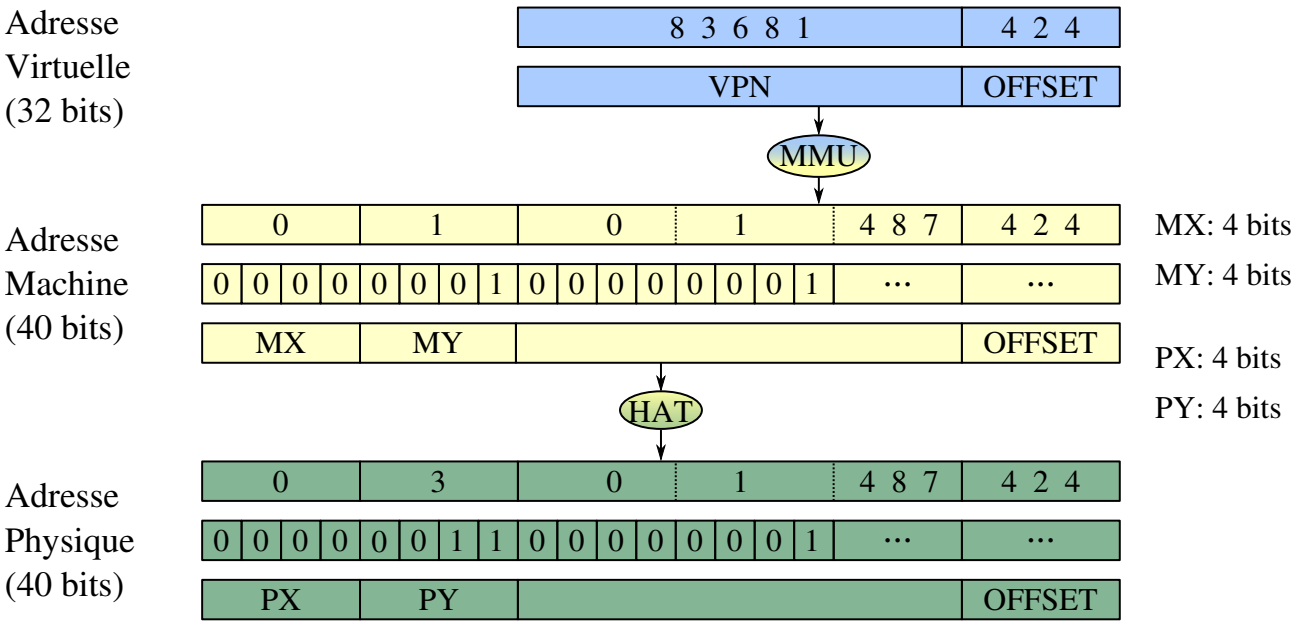

FIGURE 4.9 – Traduction d'une adresse machine en adresse physique pour un accès interne avec un système d'exploitation invité 40 bits

4.2.3 Accès externes

L'hyperviseur n'est pas impliqué dans les accès que fait la machine virtuelle aux périphériques externes. La différence entre un accès interne et un accès externe est faite *via* une recherche dans une table nommée *DevAccess*. Si l'adresse machine correspond à une entrée dans la table, alors le *Hardware Address Translator* agit comme un mécanisme de segmentation. C'est le cas si le périphérique ciblé par l'adresse est effectivement alloué à la machine virtuelle. Cette table contient plusieurs entrées où chacune contient deux d'informations :

– L'adresse physique du segment associé au périphérique ;
– Le complément à deux de la taille, en octets, de ce segment.

Nous utilisons le complément à deux de la taille plutôt que la taille pour réduire le matériel nécessaire à la vérification des accès externes.

La figure 4.10 présente un exemple du fonctionnement d'un *Hardware Address Translator* pour faire un accès à un périphérique.

La machine virtuelle de notre exemple est déployée sur 4 clusters, et les champs *MX* et *MY* ont une largeur de 1 bit. L'adresse machine émise par le cœur est 0xA8100060. Cette adresse est alors masquée par le complément à deux contenu dans la table et est ensuite comparée avec les adresses physiques de base des segments de tous les périphériques alloués à la machine virtuelle. Si une des comparaisons est vraie, alors l'adresse machine est valide et la requête est envoyée au périphérique ciblé. Si par contre aucune comparaison n'est vraie, alors la requête est considérée comme un accès

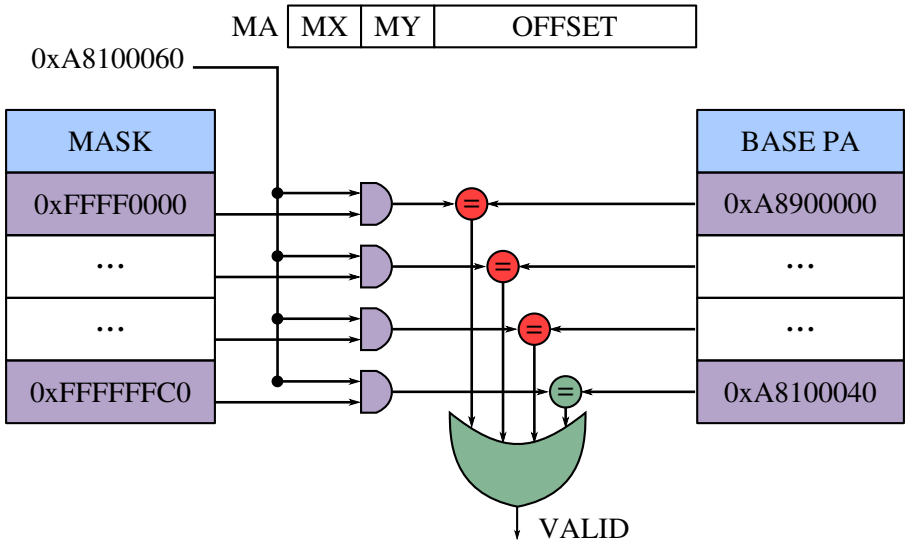

FIGURE 4.10 – Mécanisme de vérification d'un accès externe

interne et elle est donc traduite avec une adresse physique contenue dans les clusters alloués à la machine virtuelle. Si cette adresse n'est pas valide, i.e. qu'elle cible un périphérique non alloué à la machine virtuelle, elle va être acheminée à la cible par défaut du cluster, et une erreur sera retournée au cœur sous forme de *bus error*. Cette erreur indique au système d'exploitation que le cœur a essayé d'accéder à une adresse non existante.

4.2.4 Utilisation de tables de segment pour les accès internes

Le composant *Hardware Address Translator* utilise deux mécanismes différents pour fournir des traductions : par cluster dans le cas d'un accès interne, et par segment dans le cas d'un accès ciblant un périphérique externe. Ce choix a été fait dans le but de garder les *Hardware Address Translators* les plus petits possibles et donc de minimiser le surcoût matériel de notre solution. Une autre solution, plus coûteuse, consisterait à utiliser le mécanisme basé sur les segments pour les deux types d'accès. Le principal avantage d'une telle solution serait l'augmentation de la flexibilité au niveau de l'allocation des clusters, qui n'auraient plus besoin d'être contigus et de forme convexe. En contrepartie, outre la phase de configuration plus longue, le principal désavantage serait le coût matériel. En effet, le nombre d'entrées dans un composant *Hardware Address Translator* dépendrait du nombre maximal de clusters potentiellement allouables à une machine virtuelle. Plus précisément, les *Hardware Address Translators* contiennent actuellement K entrées de 10 octets pour les périphériques externes, ce qui est donc de taille fixe. En pratique, une valeur de K égale à 5 suffit pour les périphériques externes considérés. Si nous voulons utiliser une méthode basée uniquement sur des segments, cela requiert $K + N \times 5$ entrées de 10 octets, où N représente le nombre maximal de clusters allouables à une machine virtuelle et 5 est le nombre d'entrées nécessaires par cluster ; en effet, il faut une entrée par cluster pour le segment de RAM, une pour le DMA, une pour l'XICU, une pour le cryptoprocesseur et une pour le composant SVM Agent.

4.2.5 Isolation de l'hyperviseur

Pour permettre l'isolation de l'hyperviseur dans son cluster dédié nous avons implémenté des *Hardware Address Translators* particuliers dans le cluster hyperviseur, qui ne possèdent pas d'interface de configuration. Autrement dit, la configuration concernant l'isolation de l'hyperviseur est fondue dans les composants et il est impossible de venir adresser ces composants, de part l'absence de port VCI cible de configuration. La configuration des *Hardware Address Translators* hyperviseur cantonne l'hyperviseur dans le cluster $(0, 0)$ de la plateforme. L'hyperviseur a aussi accès à certains périphériques externes, tels que le TTY ou la BROM. L'hyperviseur peut néanmoins venir adresser n'importe quel XICU de la plateforme, afin de pouvoir réveiller les processeurs des machines virtuelles lorsque l'hyperviseur veut démarrer une nouvelle machine virtuelle. Nous sommes conscients que ce comportement pourrait mener à un déni de service sur les XICUs des machines virtuelles. Des modifi-

cations matérielles, au sein des XICUs, pourraient être apportées permettant de pallier ce problème, mais n'ont pas été implémentées au cours de cette thèse.

Nous avons conscience que la présence d'un *Hardware Address Translators* Hyperviseur, dont la configuration est fondue dans le composant, présente une perte de généricité pour la plateforme. En effet, la plateforme n'est pas utilisable sans hyperviseur, ou alors dans un mode dégradé où le cluster $(0, 0)$ est inutilisable. Une solution logicielle pourrait pallier cette perte de généricité, en utilisant les mêmes HATs que dans les clusters des machines virtuelles et en mettant en place une procédure logicielle effectuant la configuration de ces composants pour le cluster hyperviseur au démarrage de la machine. La mise en place de cette solution demanderait une vérification du code mettant en place la configuration des *Hardware Address Translators* de l'hyperviseur.

4.2.6 Conclusion

Nous avons présenté dans cette section notre solution pour l'isolation des machines virtuelles. Pour cela, nous avons ajouté un troisième espace d'adressage permettant la virtualisation complète et donc l'exécution de systèmes d'exploitation invités non modifiés, ainsi que des composants matériels en charge de réaliser les traductions des adresses depuis ce troisième niveau d'adressage. De plus, ces composants permettent de cantonner les machines virtuelles et l'hyperviseur dans les clusters qui leur ont été alloués. Pour réaliser ces traductions, les *Hardware Address Translators* utilisent des informations de topologie pour les accès aux périphériques répliqués – typiquement de type RAM, DMA ou XICU –, et des tables de segments pour les accès aux périphériques externes. L'utilisation de ces deux mécanismes distincts permet de réduire le coût matériel des *Hardware Address Translators* au prix d'une perte de flexibilité sur l'allocation des clusters pour les machines virtuelles. En définitive, le coût matériel d'un composant *Hardware Address Translator* reste faible (182 octets de mémoire), et la latence ajoutée pour traduire une adresse machine est de deux cycles, dont un cycle pour le mécanisme de traduction.

4.3 Partage des périphériques

Dans notre travail, le partage des périphériques est géré par la méthode d'assignation directe. Nous offrons des périphériques multi-canaux, où chaque machine virtuelle se voit attribuer un canal exclusif. Les HATs permettant de restreindre les accès des machines virtuelles, nous sommes donc assurés qu'une machine virtuelle ne peut pas atteindre un canal qui ne lui a pas été attribué. Cependant, certains périphériques externes possèdent une capacité *DMA*, autrement dit ces composants peuvent initier des accès mémoire. Il est donc nécessaire de placer des HATs devant de tels périphériques, pour deux raisons. D'une part pour des raisons de sécurité, car nous ne souhaitons pas qu'un

périphérique d'une machine virtuelle puisse venir lire ou écrire le contenu d'une zone mémoire attribuée à une autre machine virtuelle. D'autre part, car les systèmes d'exploitation des machines virtuelles fournissent des adresses machine aux périphériques, et il est donc nécessaire d'avoir des HATs permettant de traduire ces adresses machine en adresses physiques. Pour cela, nous avons dû développer un *Hardware Address Translator* multi-canaux pour pouvoir traduire les requêtes des périphériques externes. Ce composant doit être multi-canaux car les requêtes des périphériques peuvent être issues de plusieurs machines virtuelles différentes et donc ne nécessitent pas la même traduction. Ainsi ce HAT doit fournir la bonne traduction en fonction de la machine virtuelle qui effectue la requête. Ce composant est présenté dans la sous-section 4.3.3. Deux composants ont dû être développés : le composant MULTI_TTY_VT présenté dans la sous-section 4.3.1 et le composant MULTI_IOC présenté dans la sous-section 4.3.2. Enfin, il a été nécessaire de trouver un mécanisme permettant de rediriger les interruptions matérielles des canaux des périphériques vers les XICUs des machines virtuelles associées à ces canaux. Ce mécanisme est présenté dans la sous-section 4.3.4.

Il est important de préciser que le composant MULTI_TTY_, présenté dans la sous-section 4.3.1, a été développé à cause de notre environnement de simulation. En effet ce composant nous permet d'avoir une interface utilisateur-hyperviseur et un accès aux machines virtuelles simples. Dans un environnement réel l'utilisateur doit communiquer avec l'hyperviseur et les machines virtuelles par le biais du réseau. L'utilisation du composant MULTI_TTY_VT nous permet donc d'éviter de porter une pile réseau dans notre hyperviseur et dans les systèmes d'exploitation que nous exécutons dans les machines virtuelles. Néanmoins les concepts présentés, à savoir l'allocation de canaux exclusifs pour les machines virtuelles et notre mécanisme de gestion des interruptions, restent vrais et seraient utilisés à l'identique dans le cas d'un contrôleur réseau.

4.3.1 Composant MULTI_TTY_VT

Dans l'architecture conçue, chaque machine virtuelle, ainsi que l'hyperviseur, possède un accès exclusif à un ou plusieurs terminaux virtuels du TTY. Chaque terminal, ou canal, du TTY, contient un ensemble de terminaux virtuels. Pour cela, nous avons dû modifier le composant TTY existant pour offrir ce service de terminaux virtuels. Ce nouveau composant, nommé MULTI_TTY_VT, a été réalisé par Jean-Baptiste Bréjon au cours de son stage de fin d'étude. Il permet de permuter l'affichage d'un terminal (fenêtre XTerm) entre plusieurs terminaux virtuels. Il peut y avoir entre 1 et N terminaux, où chaque terminal contient entre 1 et M terminaux virtuels. M est limité par la plage d'adresses qui a été attribuée, dans la plateforme, au composant MULTI_TTY_VT.

Comme le montre la table 4.2, nous avons fixé le nombre de terminaux à 5, un étant réservé pour l'hyperviseur et les 4 autres pour les machines virtuelles. Le nombre de terminaux virtuels correspond au nombre maximum de machines virtuelles pouvant être exécutées simultanément.

	Terminal 0	Terminal 1	Terminal 2	Terminal 3	Terminal 4
VT_0	Hyperviseur	Vide	Vide	Vide	Vide
VT_1	Vide	$Term_0\ VM_0$	$Term_1\ VM_0$	$Term_2\ VM_0$	$Term_3\ VM_0$
VT_2	Vide	$Term_0\ VM_1$	$Term_1\ VM_1$	$Term_2\ VM_1$	$Term_3\ VM_1$
...	...	...	...	...	...
VT_N	Vide	$Term_0\ VM_N$	$Term_1\ VM_N$	$Term_2\ VM_N$	$Term_3\ VM_N$

TABLE 4.2 – Table de la répartition des différents terminaux et terminaux virtuels

La figure 4.11 présente l'organisation du segment de l'espace d'adressage attribué au MULTI_TTY_VT, et l'ensemble des registres adressables du composant. Les canaux contenus dans le segment $VTerm_0$ (terminal virtuel 0) sont réservés à l'hyperviseur, les autres segments $VTerm_i$ sont attribués aux machines virtuelles. Chaque $VTerm$ contient 5 canaux (ou terminaux), et à chaque canal est associé un fichier Unix dans lequel on stockera les informations écrites par les machines virtuelles dans ce terminal. Les registres SEL_I sont accessibles seulement par l'hyperviseur et lui permettent de basculer l'affichage d'un $VTerm$ dans un terminal. Par exemple, mettre la valeur 1 dans SEL_1 permet d'afficher le $VTerm_1$, ce qui correspond au contenu du terminal 0 de la machine virtuelle 0.

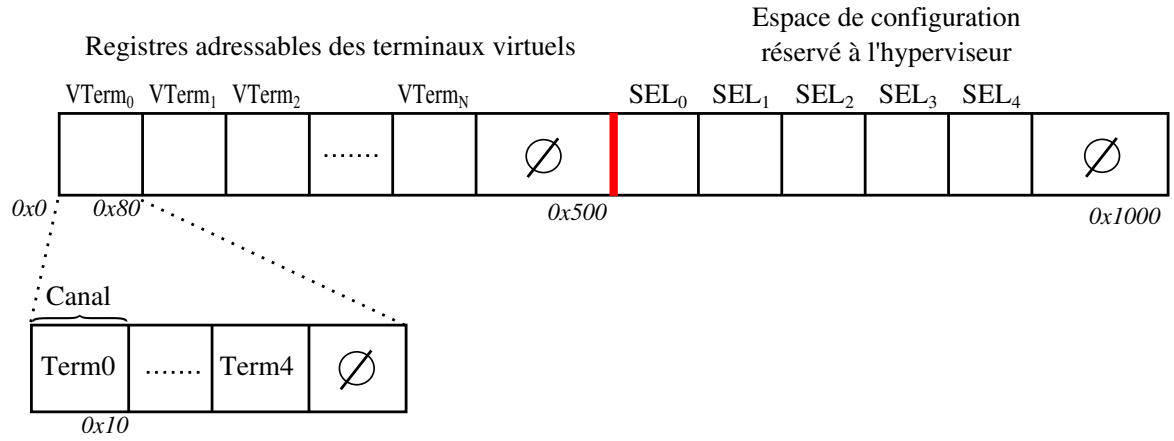

FIGURE 4.11 – Segment mémoire du composant MULTI_TTY_VT

Pour des raisons pratiques, chaque terminal virtuel d'un terminal du TTY se voit attribué un fichier Unix. Par exemple, le $VTerm_1$ du terminal 1 possède un fichier nommé *Term1_1.txt* et contient les informations du terminal 0 de la machine virtuelle 0. Dans le nom des fichiers Unix, le premier numéro détermine à quel terminal ils sont associés et le second numéro représente le numéro du terminal virtuel et donc à quelle machine virtuelle ils appartiennent. La figure 4.12 représente le mécanisme de sélection d'un fichier Unix, ici le fichier *Term1_1.txt*. Les deux fichiers représentés sur la figure sont associés au terminal 1, leur affichage sur le terminal 1 est donc contrôlé par la valeur du registre SEL_1. Sur la figure 4.12, le registre SEL_1 est à la valeur 1 ce qui signifie que ce sont les informations de la machine virtuelle 0 qui sont affichées sur ce terminal, autrement dit le fichier *Term1_1.txt*.

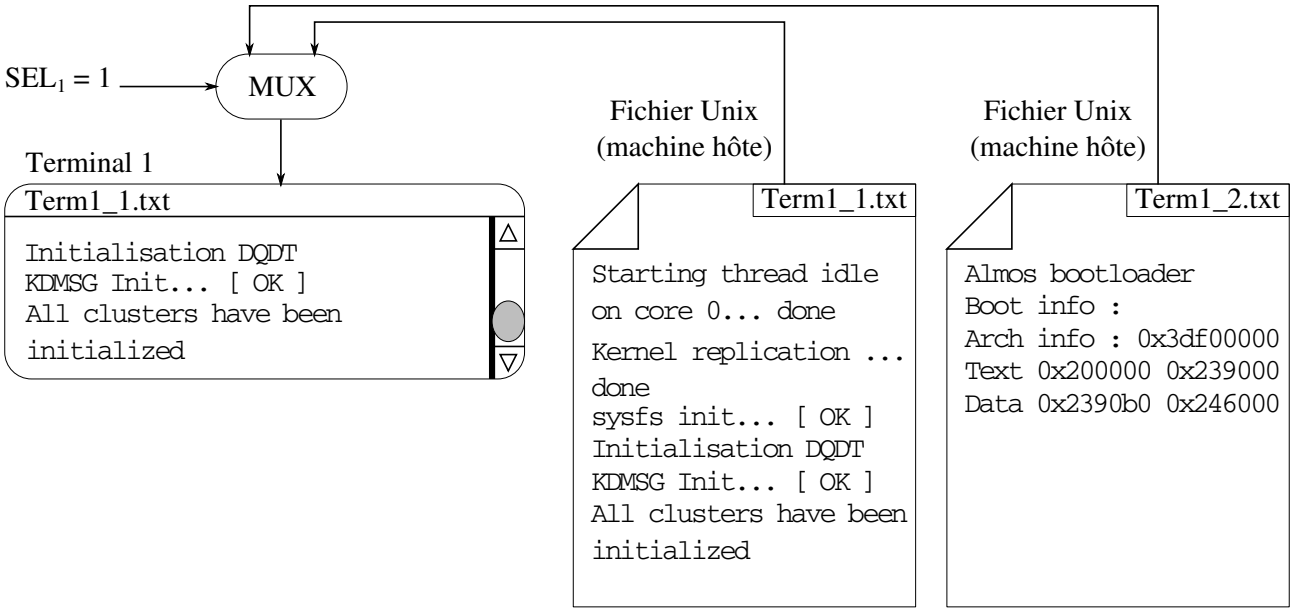

FIGURE 4.12 – Exemple de sélection de fichier par le composant MULTI_TTY_VT

Le multiplexage des terminaux est réalisé grâce aux registres SEL_I. L'hyperviseur implémente une fonction réalisant ce changement d'affichage, et celle-ci est nommée `switch_display`. Cette fonction prend en argument un numéro de terminal virtuel. Elle fait appel à la fonction `tty_switch` du pilote du composant MULTI_TTY_VT ainsi qu'à la fonction `iopic_switch`. La figure 4.13 permet d'illustrer ce mécanisme de changement d'affichage. La fonction `tty_switch` permet de modifier l'affichage des terminaux de la machine virtuelle 0 vers la machine virtuelle 2. La fonction `iopic_switch` permet de rediriger les interruptions des terminaux vers l'XICU en charge de ces interruptions, c'est-à-dire celle de la machine virtuelle 2. Le mécanisme de gestion des interruptions est expliqué en détails dans la sous-section 4.3.4.

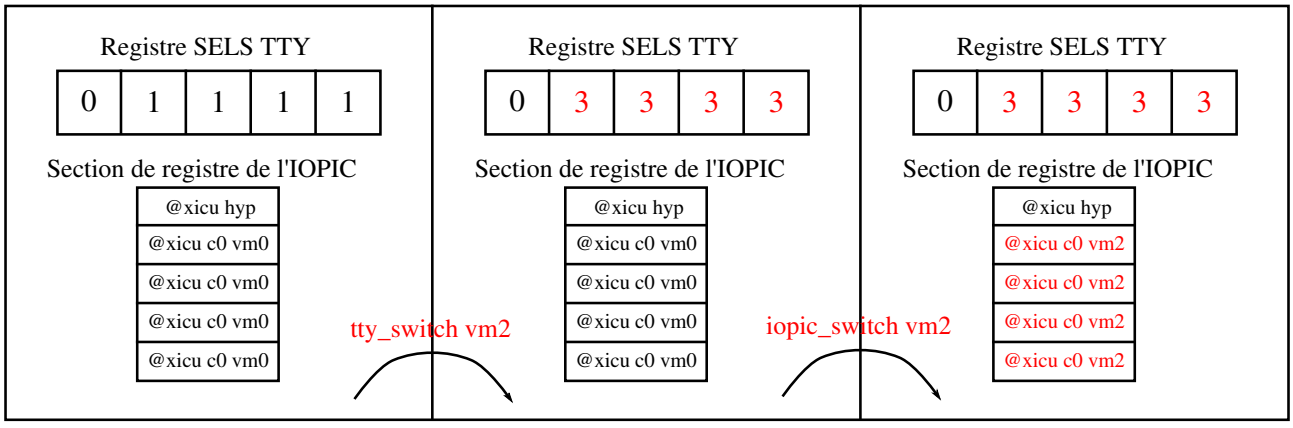

FIGURE 4.13 – Exécution de la fonction `switch_display`

La figure 4.14 présente l'architecture interne du composant MULTI_TTY_VT. Comme expliqué précédemment ce composant est composé de plusieurs canaux. Chaque canresponsable charge d'un

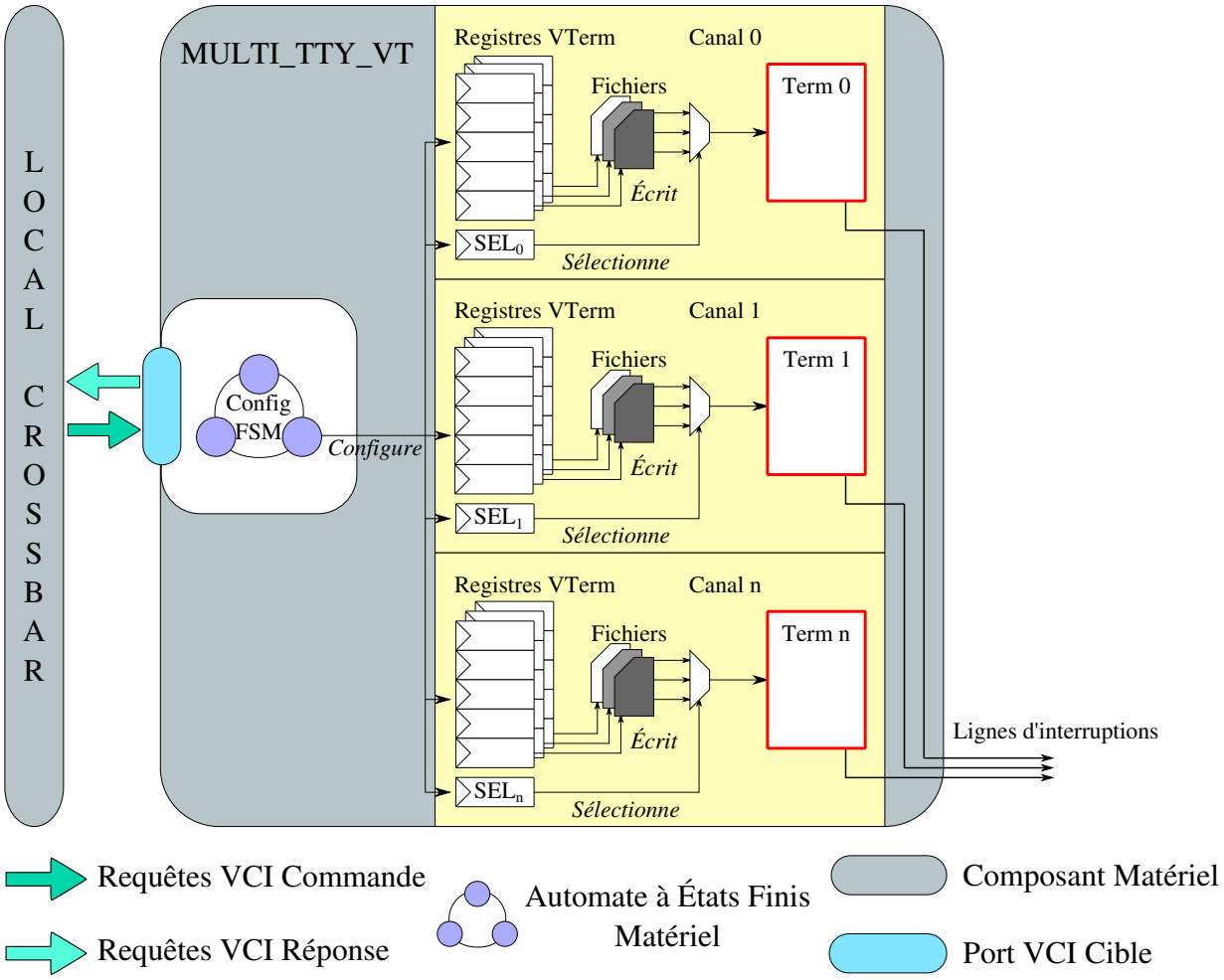

FIGURE 4.14 – Architecture interne du MULTI_TTY_VT

terminal et contient plusieurs ensembles de registres associé à chaque *VTerm*. Chaque ensemble de registres `VTERM`, attribué à une machine virtuelle, permet d'écrire dans le fichier associé. Le terminal associé au canal affiche le contenu du fichier sélectionné par la valeur du registre `SEL` du canal.

4.3.2 Composant MULTI_IOC

Comme présenté précédemment, chaque machine virtuelle se voit attribuée un accès exclusif à un canal du contrôleur de disque. Ainsi, une machine virtuelle se voit interdire tout accès aux canaux des autres machines virtuelles s'exécutant sur la plateforme. De plus, chaque machine virtuelle possède son propre disque. Pour offrir ce service, il a fallu développer un nouveau composant, nommé MULTI_IOC. Ce composant est une adaptation "multi-canaux" du composant IOC [89] déjà existant. Les modifications apportées à ce composant concernent non seulement les canaux, mais aussi la gestion de plusieurs disques.

La figure 4.15 présente l'architecture du composant MULTI_IOC. Ce composant possède une capacité DMA permettant d'initier des transferts vers la mémoire. Pour cela, il est composé d'un port VCI initiateur. Le MULTI_IOC possède aussi un port VCI cible lui permettant de recevoir les différentes commandes de configuration en provenance des machines virtuelles. Ces configurations sont gérées par l'automate `CONFIG_FSM`. Lorsque ce dernier reçoit les commandes, il détecte le canal ciblé par la configuration et écrit dans les différents registres du canal ciblé. L'automate `CMD_FSM` est en charge d'émettre les requêtes d'écriture ou de lecture vers la mémoire. Cet automate est au service des différents canaux du composant. Pour cela, il utilise une politique *Round-Robin* pour servir les différents canaux demandeurs. Les canaux du composant MULTI_IOC contiennent un ensemble de registre permettant d'initier des opérations de lecture ou d'écriture sur le disque associé au canal, ainsi qu'un automate `CH_FSM`. Cet automate est en charge d'accéder au disque associé au canal et d'effectuer une demande à l'automate `CMD_FSM` pour venir lire ou écrire en mémoire. Chaque canal possède aussi une ligne d'interruption reliée au composant IOPIC, permettant ainsi de signaler à la machine virtuelle associée au canal que celui-ci a terminé l'opération demandée.

Ce composant permet aux machines virtuelles d'accéder simultanément à leur disque. En revanche, l'émission ou la réception de données vers la mémoire est sérialisée par l'automate `CMD_FSM`, et par la présence d'un unique port VCI initiateur. Ceci peut induire un point de contention lorsque plusieurs machines virtuelles décident d'effectuer des opérations sur leur disque. Cependant, il est important de noter que le surcout, en termes de cycles, reste minime car le temps de transfert d'un bloc du disque vers la mémoire est très inférieur à la latence d'accès d'un disque. Or, les disques peuvent être accédés de façon parallèle, ce qui implique que le surcoût induit par la présence d'un seul port initiateur reste peu pénalisant.

Les registres adressables, répliqués par canal, du composant MULTI_IOC sont présentés dans la table 4.3.

Nom du registre	Permission d'accès	Taille (en bits)
`BLK_DEV_BUFFER`	Écriture / Lecture	32
`BLK_DEV_BUFFER_EXT`	Écriture / Lecture	8
`BLK_DEV_LBA`	Écriture / Lecture	32
`BLK_DEV_COUNT`	Écriture / Lecture	32
`BLK_DEV_OP`	Écriture	2
`BLK_DEV_STATUS`	Lecture	4
`BLK_DEV_IRQ_ENABLE`	Écriture / Lecture	1
`BLK_DEV_SIZE`	Lecture	32
`BLK_DEV_BLOCK_SIZE`	Lecture	32

TABLE 4.3 – Table des registres adressables du MULTI_IOC

`BLK_DEV_BUFFER` : ce registre contient les 32 bits de poids faible de l'adresse source ou destination de

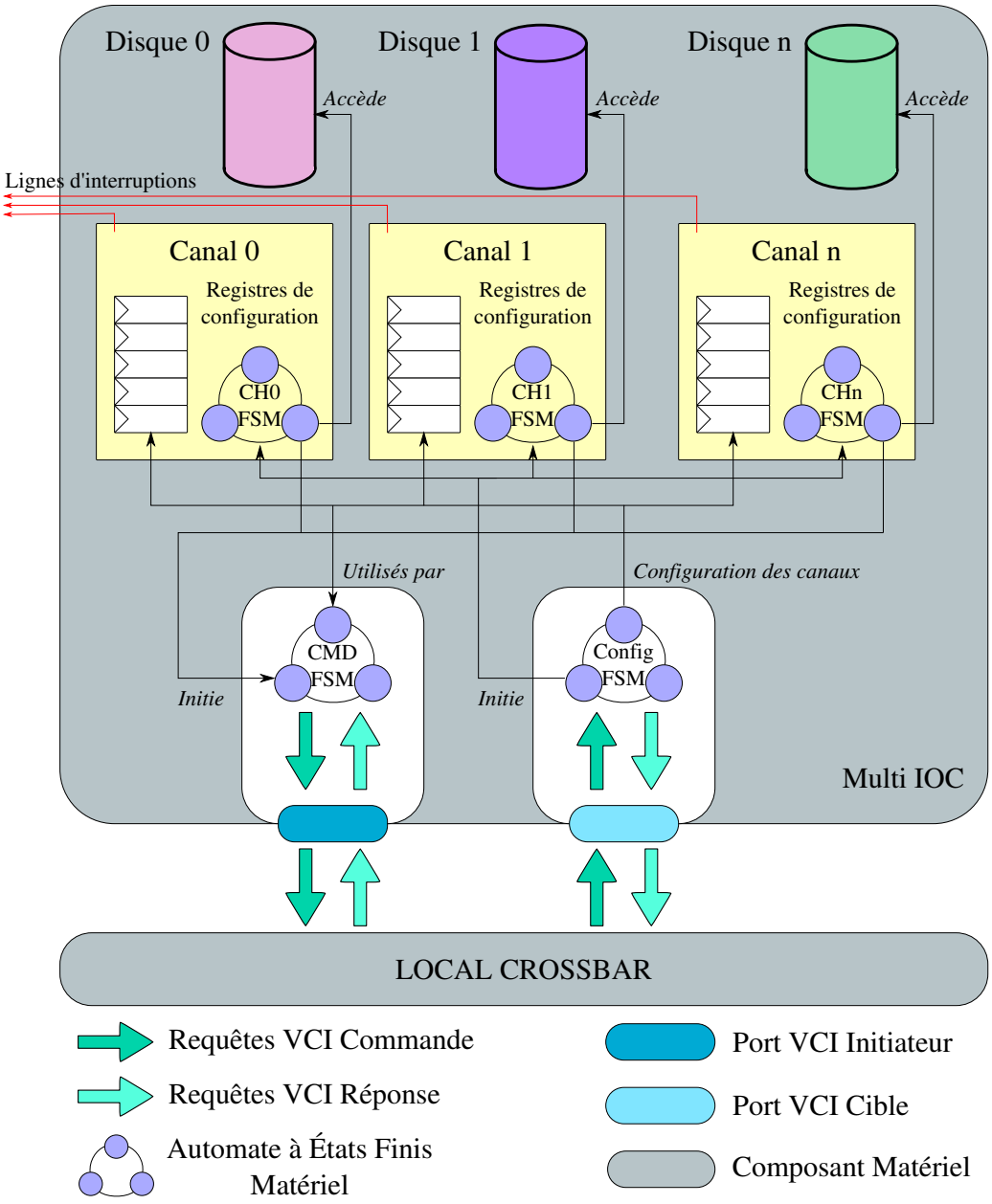

FIGURE 4.15 – Architecture interne du MULTI_IOC

la requête (selon l'opération).

`BLK_DEV_BUFFER_EXT` : ce registre contient les 8 bits d'extension de l'adresse source ou destination de la requête.

`BLK_DEV_LBA` : ce registre contient le *Logical Block Address*, ou numéro de secteur, ciblé par la requête.

`BLK_DEV_COUNT` : ce registre contient la taille de la requête en nombre de blocs.

`BLK_DEV_OP` : une écriture à cette adresse permet d'informer le contrôleur de l'opération qui doit être réalisée, une lecture ou une écriture d'un bloc. Ce registre permet d'initier le transfert.

`BLK_DEV_STATUS` : ce registre contient le statut du canal, il peut contenir 5 valeurs : *BLK_IDLE* signifiant que le canal est disponible, *BLK_BUSY* signifiant que le canal est en cours de fonctionnement, *BLK_READ_SUCCESS* signifiant qu'une opération de lecture s'est bien réalisée, *BLK_READ_ERROR* signifiant qu'une erreur est survenue pendant une opération de lecture, *BLK_WRITE_SUCCESS* signifiant qu'une opération d'écriture s'est terminée sans erreur, *BLK_WRITE_ERROR* signifiant qu'une erreur est survenue pendant une opération d'écriture.

`BLK_DEV_IRQ_ENABLE` : ce booléen permet d'activer ou non la génération d'une interruption matérielle à la fin du traitement d'une requête.

`BLK_DEV_SIZE` : ce registre contient la taille totale du disque dur associé au canal, en nombre de blocs.

`BLK_DEV_BLOCK_SIZE` : ce registre contient la taille d'un bloc en octets.

Le registre qui pose problème est le registre `BLK_DEV_BUFFER`. L'adresse fournie par le système d'exploitation est une adresse machine. Dans le cas d'une lecture, le composant se sert de cette adresse pour écrire dans la mémoire. Or, cette adresse n'a de sens que derrière un *Hardware Address Translator*. Avec la traduction fournie par un *Hardware Address Translator* les adresses émises par le disque seront redirigées vers les clusters de la machine virtuelle qui a initiée la transaction. Cependant le composant MULTI_IOC est multi-canaux. Un canal étant affecté à une machine virtuelle, il est donc nécessaire que le *Hardware Address Translator* devant le disque soit capable de fournir une traduction différente pour chaque canal.

4.3.3 Composant MULTI_HAT

Le composant MULTI_HAT est un *Hardware Address Translator* multi-canaux qui a pour but de répondre au problème posé par le fonctionnement des périphériques externes possédant une capacité DMA. Le MULTI_HAT possède le même nombre de canaux que le périphérique externe derrière lequel il se situe, typiquement le nombre maximal de machines virtuelles pouvant être exécutées simultanément. Pour chaque canal, ce composant possède un ensemble de registres. Cet ensemble de registres est configuré par l'hyperviseur lors du déploiement d'une machine virtuelle. Ce *Hardware Address Translator* multi-canaux est configuré de la même façon que les HATs présents devant les cœurs alloués à la machine virtuelle.

La figure 4.16 présente le fonctionnement du MULTI_IOC couplé avec le MULTI_HAT. Lorsqu'un système d'exploitation d'une machine virtuelle désire utiliser le contrôleur de disque, celui-ci doit

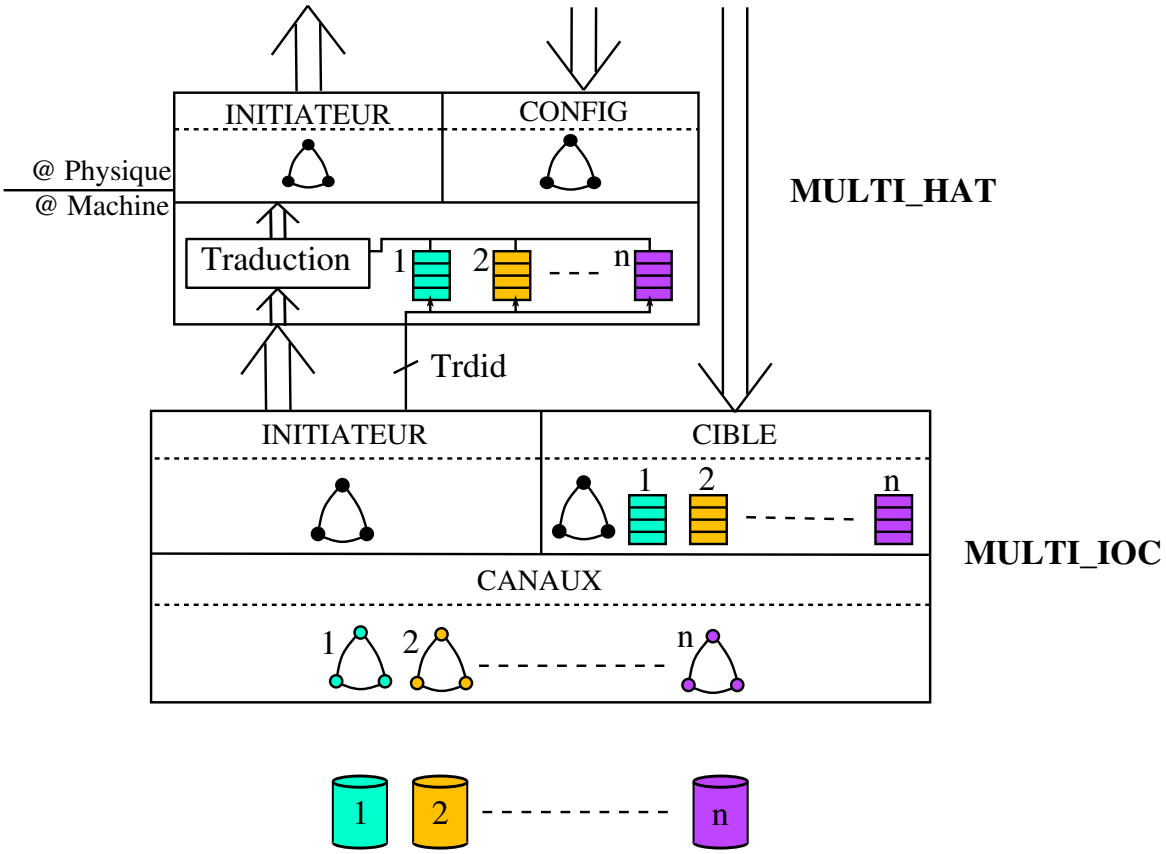

FIGURE 4.16 – Fonctionnement d'un MULTI_HAT couplé à un MULTI_IOC

fournir l'adresse à laquelle les données doivent être lues ou écrites par le contrôleur. Pour cela, il est nécessaire de configurer le registre `BLK_DEV_BUFFER` du composant MULTI_IOC mais la machine virtuelle, s'exécutant dans l'espace machine, va donc donner au contrôleur une adresse machine et non une adresse physique. En traitant la requête de la machine virtuelle, le contrôleur de disque va alors émettre une adresse machine qui sera par la suite traduite par le MULTI_HAT situé derrière le composant MULTI_IOC.

Sur la figure 4.16, la machine virtuelle 1 utilise le canal 1 du contrôleur de disque. Lorsque l'automate initiateur traite la requête de la machine virtuelle 1, celui-ci envoie un message vers le MULTI_HAT qui doit traduire l'adresse reçue en utilisant l'ensemble de registre associé à la machine virtuelle 1. Pour cela, le contrôleur de disque émet dans le champ *TRDID* de la requête VCI le numéro d'instance de la machine virtuelle pour qui la requête est effectuée. Ainsi, à la réception de la requête, le composant MULTI_HAT utilise le champ *TRDID* pour savoir quel ensemble de registres doit être utilisé pour réaliser la traduction.

4.3.4 Gestion des interruptions

Chaque périphérique externe possède une ligne d'interruption par canal. Par exemple, dans le cas du composant MULTI_TTY_VT, une interruption est générée à chaque appui sur une touche dans un terminal. Le composant MULTI_TTY_VT possède donc autant de fils d'interruptions que de terminaux. Il faut donc pouvoir rediriger les interruptions vers les différentes machines virtuelles. Pour cela, on utilise un composant nommé IOPIC (*Input Output Programmable Interrupt Controller*). Ce composant permet de traduire une interruption matérielle *HardWare Interrupt* (HWI) en interruption logicielle *Write-Triggered Interrupt* (WTI). Autrement dit, les interruptions sortantes d'un canal d'un périphérique sont routées vers l'IOPIC, qui déclenche ensuite une interruption logicielle à l'XICU située dans un des clusters alloués à la machine virtuelle. L'XICU cible convertit finalement cette interruption logicielle en interruption matérielle vers un processeur.

L'IOPIC est un composant programmable, l'hyperviseur peut donc changer dynamiquement l'adresse du contrôleur d'interruption à laquelle sera envoyée la WTI. Par exemple, dans le cas du composant MULTI_TTY_VT, cela permet d'envoyer l'interruption à la machine virtuelle dont l'affichage se trouve sur le terminal au moment où la touche est appuyée. Pour cela, il est nécessaire de configurer les canaux de l'IOPIC. Chaque canal de l'IOPIC gère une entrée d'interruption HWI et chaque canal contient quatre registres :

- `IOPIC_ADDRESS`, qui contient les 32 bits de poids faible de l'adresse physique de la WTI associée au canal HWI. Typiquement ce registre contient l'adresse physique du canal WTI de l'XICU en charge de l'interruption.
- `IOPIC_EXTEND`, qui contient les 8 bits de poids fort de l'adresse physique de la WTI associée au canal HWI. Typiquement, ce registre contient les coordonnées (x, y) du cluster contenant l'XICU en charge de l'interruption.
- `IOPIC_MASK`, qui permet d'activer ou non la génération d'une interruption WTI après la réception d'une interruption HWI.
- `IOPIC_STATUS`, qui contient le statut du canal HWI de l'IOPIC. Seuls les deux bits de poids faible sont pertinents, le bit 0 donne la valeur courante de la ligne d'interruption HWI et le bit 1 permet de savoir si une erreur est survenue lors de la transaction WTI.

La figure 4.17 illustre le mécanisme de traduction d'une interruption HWI en interruption WTI. Premièrement, le périphérique lève sa ligne d'interruption HWI (1). L'IOPIC envoie alors une écriture à l'adresse physique de la WTI associée (2). Cette adresse est la concaténation du contenu des registres `IOPIC_ADDRESS` et `IOPIC_EXTEND`. Cette écriture est à destination de l'XICU en charge de l'interruption du périphérique. À la réception de l'écriture de l'IOPIC, l'XICU va lever la ligne d'interruption en charge de cette WTI au niveau du cœur (3). Le cœur va alors exécuter son gestionnaire d'interruption et venir acquitter l'interruption au sein de l'XICU en effectuant une lecture dans le canal de la WTI au niveau de l'XICU (4). Le cœur exécute alors le traitement de l'interruption puis acquitte l'interruption au niveau du périphérique (5). Une fois le périphérique acquitté, celui-ci baisse

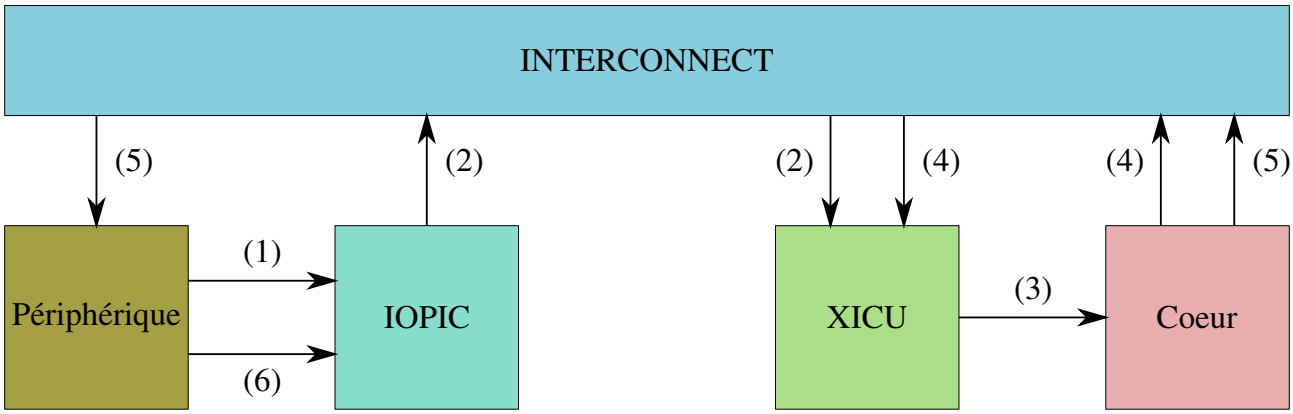

FIGURE 4.17 – Mécanisme de traduction d'une interruption HWI en interruption WTI

sa ligne d'interruption HWI au niveau de l'IOPIC (6).

4.3.5 Conclusion

Nous avons présenté dans cette section notre solution concernant le partage des périphériques entre les machines virtuelles. Celle-ci consiste à utiliser le principe d'assignation directe et d'offrir aux machines virtuelles des canaux de périphériques exclusifs. Cette méthode, couplée à la présence des HATs, permet d'assurer qu'aucune machine virtuelle, ni même l'hyperviseur, ne puisse accéder aux canaux des périphériques qui ont été alloués à une autre machine virtuelle. Pour cela, nous avons dû concevoir deux nouveaux périphériques, le MULTI_TTY_VT et le MULTI_IOC, car les composants existants ne permettaient pas d'offrir ce service d'exclusivité d'un canal. De plus, nous avons dû développer un nouveau type de *Hardware Address Translator* possédant plusieurs canaux, et donc plusieurs configurations de traduction, afin de pouvoir gérer les requêtes émanant des périphériques externes multi-canaux. Enfin, cette solution utilise le composant IOPIC pour rediriger les interruptions provenant des périphériques externes vers les XICUs des machines virtuelles correspondantes.

4.4 Mécanisme de démarrage d'une machine virtuelle

La procédure de démarrage d'une machine virtuelle peut être divisée en 7 étapes :
- (1) Allocation des clusters ;
- (2) Allocation des canaux des périphériques externes et routage des interruptions ;
- (3) Génération de la description de l'architecture de la machine virtuelle (*device-tree*) ;
- (4) Réveil des cœurs alloués : cette étape doit être faite par l'hyperviseur en envoyant une WTI ;
- (5) Configuration des composants en charge de l'isolation des machines virtuelles ;
- (6) Configuration des composants en charge de l'arrêt des machines virtuelles ;
- (7) Isolation des clusters alloués.

La difficulté de cette procédure de démarrage est que l'hyperviseur s'exécutant sur un autre cluster ne peut pas agir sur le cluster cible de la machine virtuelle hormis l'envoi d'une WTI permettant le réveil des cœurs. Néanmoins, nous ne voulons pas démarrer une machine virtuelle dans un environnement non isolé. Nous devons donc nous assurer qu'un mécanisme d'isolement est mis en œuvre entre le moment où les cœurs sont réveillés et l'exécution de la première instruction du code de la machine virtuelle.

Pour résoudre cette difficulté, la procédure de démarrage fait en sorte que chaque cœur contenu dans les clusters alloués à la future machine virtuelle exécute un code hyperviseur permettant l'isolation de celle-ci. Ce code doit effectuer la configuration et l'activation du *Hardware Address Translator* du cœur qui l'exécute ; il est appelé `startup_code`. Autrement dit, la future machine virtuelle n'est pas isolée par l'hyperviseur, mais s'isole de l'intérieur.

La figure 4.18 représente les différentes étapes de la procédure de démarrage d'une machine virtuelle effectuées par le cœur hyperviseur et le cœur de démarrage de la machine virtuelle. Cette procédure de démarrage a été réalisée avec l'aide de Jean-Baptiste Bréjon lors de son stage de fin d'étude.

4.4.1 Requête de l'utilisateur

La première étape consiste en la demande d'un utilisateur à l'hyperviseur de déployer une nouvelle machine virtuelle par le biais du terminal. L'utilisateur spécifie quel système d'exploitation il désire démarrer, le numéro d'instance de la machine virtuelle, ainsi que le nombre de clusters désiré. L'appel à la commande de démarrage d'une machine virtuelle depuis le terminal, donc depuis le mode utilisateur de l'hyperviseur, fait un appel système permettant de basculer dans le mode noyau de l'hyperviseur. Ceci permet de s'assurer qu'un *buffer overflow* dans le terminal ne vienne pas corrompre des données critiques de l'hyperviseur.

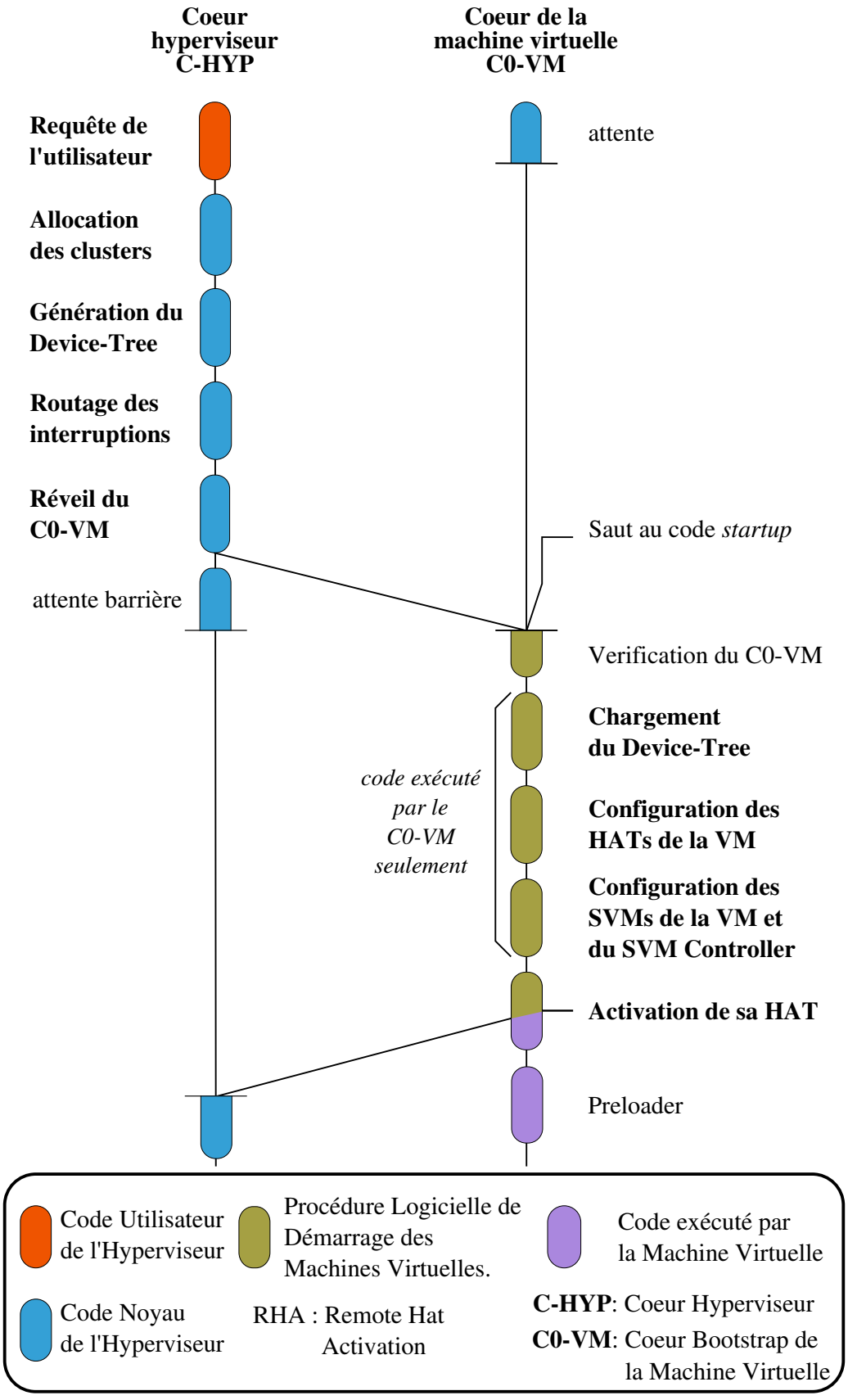

FIGURE 4.18 – Procédure de Démarrage d'une Machine Virtuelle

Une fois dans le mode noyau de l'hyperviseur, celui-ci vérifie la possibilité de lancement de cette machine virtuelle. L'hyperviseur vérifie entre autres que le nombre de cluster demandés n'est pas supérieur au nombre de clusters disponibles, que le système d'exploitation demandé est supporté, et que ce dernier n'est pas déjà en train de fonctionner dans une machine virtuelle. Pour effectuer cette vérification, on attribue un numéro d'instance à chaque machine virtuelle, qui correspond au numéro du disque comportant le code et le système de fichier de la machine virtuelle en question. Lorsque l'hyperviseur démarre une machine virtuelle, il marque ce numéro d'instance comme étant en fonctionnement et ne peut relancer la même machine virtuelle simultanément.

Actuellement la séparation de l'hyperviseur en deux niveaux de privilège, mode utilisateur et mode noyau, n'est pas implémentée et l'ensemble de l'hyperviseur fonctionne en mode noyau. Autrement dit, le mécanisme d'appel système n'a pas encore été mis en œuvre.

4.4.2 Allocation des clusters

Lorsque l'utilisateur fait une demande de lancement de machine virtuelle, l'hyperviseur doit allouer des clusters de la plateforme pour cette machine virtuelle. La fonction d'allocation des clusters cherche alors un ensemble de clusters convexe et contigu pour cette machine virtuelle. Nous avons fait ce choix pour éviter que les machines virtuelles n'interfèrent entre elles. En effet, étant donné que les machines virtuelles partagent le même réseau global (réseau L1/L2), une machine virtuelle générant beaucoup de trafic pourrait dégrader les performances d'une autre machine virtuelle. La figure 4.19 montre comment deux machines virtuelles n'ayant pas une allocation de clusters convexe pourraient se gêner mutuellement en termes de performance.

Ce choix présente aussi deux avantages : premièrement, cela permet de donner au système d'exploitation une topologie *2D-mesh* simple qui correspond à la réalité, et donc de respecter ses heuristiques de localité ; deuxièmement, ce choix permet de réduire les informations nécessaires pour représenter un ensemble de clusters alloués à une machine virtuelle, ce qui est important étant donné que le matériel va être impliqué dans la gestion des ressources.

L'algorithme pour l'allocation des clusters à une machine virtuelle est décrit dans la figure 4.20. Celui-ci est composé de deux parties. Dans la première partie (ligne 5), nous énumérons tous les couples de coordonnées (x, y) tels que $x \times y$ soit égal au nombre n de clusters requis. Ceci peut être réalisé de deux manières : soit en énumérant toutes les valeurs de x puis en calculant $y = n/x$ et en vérifiant que $x \times y = n$; soit en ayant une table pré-calculée contenant toutes les configurations possibles, cette table pouvant être indexée par n. Cette deuxième méthode permet de simplifier l'exploration des meilleures configurations, c'est-à-dire celles où x et y sont proches, car les distances moyennes entre routeurs sont réduites. Dans la seconde partie de l'algorithme, nous vérifions tous les placements possibles (lignes 8-9) et nous vérifions la disponibilité de ces placements (ligne 11-18).

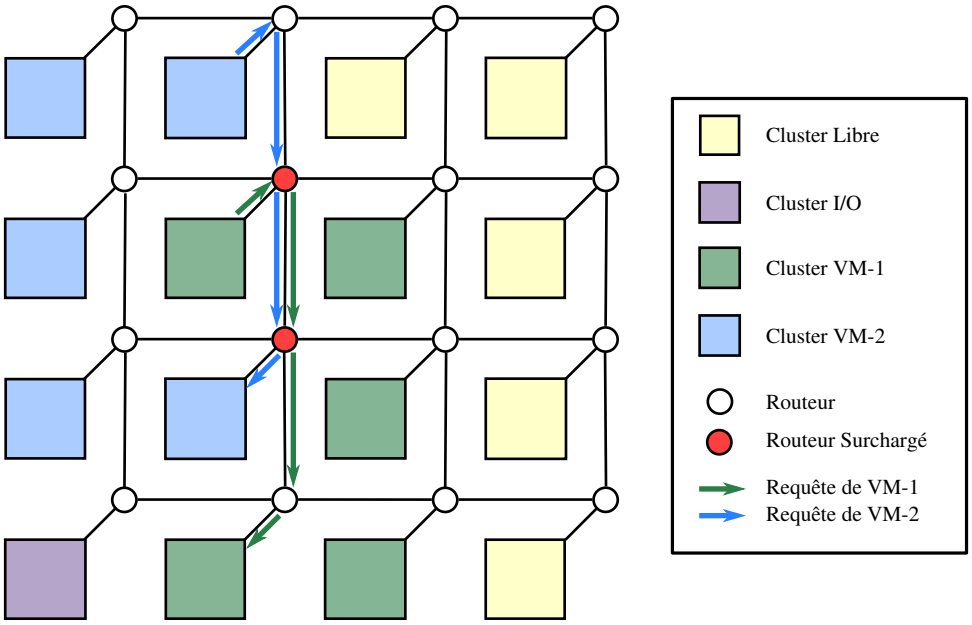

FIGURE 4.19 – Surcharge possible de routeurs suite à des allocations non convexes

Les informations de placement sont ensuite enregistrées (ligne 20-23) pour être utilisées par l'hyperviseur dans la suite de la procédure de démarrage d'une machine virtuelle.

```
1  bool allocate(int tab[X_SIZE][Y_SIZE], int n, info_s * info) {
2      if (n >= X_SIZE * Y_SIZE || n <= 0) {
3          return false;
4      }
5      for ((i, j) such that i * j = n) {
6          // Trying to find i * j free clusters
7          // on the mesh
8          for (int k = 0; k <= X_SIZE - i; k++) {
9              for (int l = 0; l <= Y_SIZE - j; l++) {
10                 // Trying corner (k, l)
11                 bool ok = true;
12                 for (int u=k; u < k+i && ok; u++) {
13                     for (int v=l; v < l+j && ok; v++) {
14                         if (tab[u][v] != 0) {
15                             ok = false;
16                         }
17                     }
18                 }
19                 if (ok) {
20                     info->x = k;
21                     info->x_size = i;
22                     info->y = l;
23                     info->y_size = j;
24                     return true;
25                 }
26             }
27         }
28     }
29     return false;
30 }
```

FIGURE 4.20 – Algorithme d'allocation des clusters. `tab` est la table des clusters utilisés, `n` le nombre de clusters requis, et `info` les informations de placement retournées.

4.4.3 Génération du Device-Tree

L'hyperviseur doit générer un fichier binaire représentant l'architecture sur laquelle la machine virtuelle sera déployée. Il doit utiliser les informations données par l'algorithme d'allocation de clusters pour construire cette représentation ainsi que le système d'exploitation demandé. Ce fichier binaire contient entre autre le nombre de clusters alloués à la machine virtuelle et leur composition, c'est-à-dire les composants contenus dans ces clusters. Plus précisément, nous allons y trouver les bancs mémoire, en précisant l'adresse de début et leur taille, les cœurs, les périphériques répliqués et les périphériques externes alloués à la machine virtuelle. Nous y trouvons aussi des informations sur la gestion des interruptions, qui permettent typiquement de savoir quelle ligne d'interruption est reliée à quel périphérique.

Le format de ce fichier de représentation de l'architecture diffère suivant le système d'exploitation demandé. Dans ce travail, nous nous sommes limités à supporter deux systèmes d'exploitation, NetBSD et ALMOS, ce dernier étant un système d'exploitation expérimental développé dans notre laboratoire. En ce qui concerne ALMOS, son format de représentation de l'architecture n'est pas standard, et est simplifié par rapport à celui requis pour NetBSD. Pour NetBSD, nous avons dû porter la librairie *Flattened Device Tree* [90, 91] au sein de l'hyperviseur. Cette librairie permet d'offrir un ensemble de fonctions permettant de construire un *Device-Tree* compatible avec le format supporté par NetBSD.

Une fois le fichier binaire construit, l'hyperviseur le stocke dans la mémoire de son cluster, puisqu'il ne peut atteindre les mémoires des autres clusters.

4.4.4 Routage des interruptions

L'hyperviseur doit configurer l'IOPIC pour permettre à la machine virtuelle de recevoir les interruptions des canaux des périphériques externes qui lui ont été alloués. Autrement dit l'hyperviseur doit configurer les canaux de l'IOPIC pour rediriger les interruptions des canaux des périphériques, qui ont été alloués à la machine virtuelle, vers les XICUs de la machine virtuelle. Ainsi les interruptions HWI générées par les canaux des périphériques seront transformées en interruptions logicielles et envoyées vers les XICUs de la machine virtuelle en charge de ces interruptions.

4.4.5 Réveil du cœur C0 de la machine virtuelle

Une fois que l'hyperviseur a configuré les interruptions pour la machine virtuelle, celui-ci envoie une WTI vers le C0-VM pour le réveiller. Le C0-VM est le cœur de démarrage de la machine virtuelle,

typiquement le cœur 0 du cluster $(0,0)$ de la machine virtuelle. Une fois réveillé, le C0-VM saute au `startup_code` pour exécuter la suite de la procédure de démarrage. Contrairement aux autres cœurs de la machine virtuelle, le C0-VM doit exécuter des fonctions d'initialisation et de configuration des périphériques alloués à la machine virtuelle, comme par exemple tous les HATs de la machine virtuelle. Les autres cœurs de la machine virtuelle seront réveillés plus tard par le `bootloader` du système d'exploitation de la machine virtuelle. Ils exécuteront aussi le `startup_code` pour activer leur HAT, mais n'effectueront pas de configuration des périphériques.

4.4.6 Chargement du Device-Tree de la machine virtuelle

Le C0-VM doit venir récupérer la description de l'architecture de la machine virtuelle générée par le cœur hyperviseur (C-Hyp) et la copier dans la mémoire de la machine virtuelle. En effet, cette description sera utilisée par la suite par le système d'exploitation de la machine virtuelle, mais celui-ci ne pourra plus accéder au cluster hyperviseur. Il faut donc que le C0-VM rapatrie au préalable, c'est-à-dire avant l'activation des HATs, cette description de l'architecture contenue dans la mémoire de l'hyperviseur. Il est impossible pour l'hyperviseur de venir la copier directement dans la mémoire de la machine virtuelle puisque celui-ci n'a accès qu'à son cluster.

4.4.7 Configuration des HATs et des SVMs

Le C0-VM doit configurer tous les HATs contenus dans les clusters alloués à la machine virtuelle ainsi que ceux présents derrière les périphériques externes. Les HATs configurés sont ceux situés au niveau des cœurs de la machine virtuelle, des DMAs, des crypto-processeurs ainsi que celui utilisé par le contrôleur de disque. Cette fonction de configuration des HATs est critique et doit être sûre, car le bon fonctionnement de l'isolation des machines virtuelles se base sur la bonne configuration des HATs. Le C0-VM utilise les informations de déploiement de la machine virtuelle pour configurer les HATs, il doit leur fournir les informations de topologie de la future machine virtuelle ainsi que l'ensemble des segments associés aux périphériques externes alloués à la machine virtuelle. Le C0-VM doit aussi fournir aux HATs la nature du système d'exploitation qui va s'exécuter : 32 ou 40 bits, puisque comme vu dans la section 4.2.2, les HATs ne fournissent pas le même service de traduction selon que le système d'exploitation est en 32 ou 40 bits.

Le C0-VM doit aussi configurer les SVMs Agent contenus dans l'ensemble des clusters alloués à la machine virtuelle ainsi que le SVM Controller pour signaler qu'une nouvelle machine virtuelle est démarrée. Ces composants sont en charge de l'arrêt des machines virtuelles et seront détaillés dans la section 4.5.1. La configuration consiste principalement à donner les informations de topologie de la machine virtuelle et son identifiant pour le SVM Controller et la topologie ainsi que la nature du système d'exploitation pour les SVMs Agent. Comme pour la fonction de configuration des HATs, la

fonction permettant de configurer les SVMs doit être sûre pour garantir le bon fonctionnement et la sécurité de l'arrêt des machines virtuelles.

4.4.8 Activation des HATs

L'activation des HATs est un moment critique au cours duquel l'isolation des machines virtuelles s'active. Chaque cœur d'une machine virtuelle vient activer son propre *Hardware Address Translator* à la fin de l'exécution du `startup_code`. Cette activation est faite via le code `Remote HAT Activation` (RHA) et comporte 6 étapes importantes. La figure 4.21 illustre la séquence complète du code RHA permettant l'activation d'un HAT.

```
1     # t0 contient les 8 bits de poids fort de l'adresse des HATs
2     # t1 contient les 32 bits de poids faible de l'adresse des HATs
3     # t2 contient le mode souhaité pour les HATs
4     # t3 contient l'adresse du point d'entrée
5     # t4 contient l'adresse de la description de l'architecture
6     # t5 contient l'adresse du TTY alloué à la machine virtuelle
7     # t6 contient l'adresse de l'IOC alloué à la machine virtuelle
8     # t7 contient l'adresse du MMC alloué à la machine virtuelle
9
10    sync
11    nop
12    mtc2 zero, $1, 0
13    mtc2 zero, $3, 0
14    mtc2 zero, $2, 0
15    move a0, t7
16    move a1, t5
17    move a2, t6
18    move a3, t4
19    mtc2 t0, $24, 2
20    sw   t2, 0(t1)
21    mtc2 zero, $24, 2
22    sync
23    nop
24    li   $24,  3
25    jalr t3
26    mtc2 $24, $1
27
```

FIGURE 4.21 – Code assembleur MIPS permettant l'activation des HATs

Tout d'abord, chaque cœur vient exécuter l'instruction `sync` permettant de synchroniser les écritures pendantes, ce qui a pour but de s'assurer que toutes les écritures liées aux opérations précédentes sont terminées (lignes 10-11).

Ensuite, le cœur désactive et efface l'intégralité de ses caches instructions et données (lignes 12-14). Ceci est fait pour s'assurer qu'aucune instruction ou donnée provenant du code hyperviseur ne sera en cache après l'isolation de la machine virtuelle, permettant ainsi d'éviter des inter-blocages liés au protocole de cohérence. En effet, ces données ou instructions seraient en cache avec les HATs

désactivés, donc en adresse physique, et lors de l'activation des HATs celles-ci seraient considérées comme étant en adresse machine. Lors d'une opération d'évincement de cache de ces lignes, elles subiraient ainsi une traduction des HATs qui n'aurait aucune correspondance dans les caches de second niveau, provoquant ainsi un dysfonctionnement matériel.

Une fois les caches désactivés puis vidés, le cœur place dans les registres réservés aux arguments des adresses (lignes 15-18) : l'adresse de la description de l'architecture, ainsi que des adresses de canaux de périphériques alloués à la machine virtuelle. Ces adresses seront utilisées par le `preloader` une fois que l'exécution du code RHA sera terminée.

L'étape suivante consiste en l'activation effective des HATs, qui se traduit par une écriture dans le registre `HAT_MODE` du composant (lignes 19-21). Cette écriture nécessite de placer au préalable les bits d'extension du cache à la valeur des 8 bits de poids fort de l'adresse des HATs (ligne 19) permettant ainsi d'effectuer une écriture à une adresse 40 bits.

Pour s'assurer que l'écriture du registre `HAT_MODE` est bien effective, il faut réaliser une instruction `sync`, permettant ainsi de garantir que les HATs sont activés et que les machines virtuelles sont isolées avant leur exécution (lignes 22-23).

Enfin la dernière étape consiste à sauter au point d'entrée du `preloader` qui sera en charge de démarrer le système d'exploitation de la machine virtuelle (ligne 25). Cette instruction est immédiatement suivie, dans le *delayed slot* du saut, par l'activation des caches instructions et données (ligne 26). Il est impossible d'effectuer ces deux instructions dans un ordre différent car sinon l'instruction `jalr` (ligne 25) serait ramenée dans le cache instructions et nous rencontrerions un problème similaire à celui évoqué précédemment.

4.4.9 Conclusion

Nous avons présenté dans cette section notre mécanisme de démarrage des machines virtuelles. Le démarrage d'une machine virtuelle est initié par l'hyperviseur mais les étapes concernant la sécurité des machines virtuelles, i.e. la configuration des HATs et leur activation, sont réalisées par les cœurs de la machine virtuelle elle-même. Ceux-ci exécutent un code nommé `startup_code`, qui permet la configuration des composants liés à la sécurité, après quoi ils exécutent le code `RHA` permettant d'activer leur HAT. Les machines virtuelles sont donc isolées de l'intérieur et seul le `startup_code` nécessite d'être vérifié.

4.5 Mécanisme d'arrêt d'une machine virtuelle

Le mécanisme d'arrêt d'une machine virtuelle vise à proposer une interface, accessible aussi bien par l'hyperviseur que par la machine virtuelle elle-même. Cette procédure assure qu'à sa complétion, la machine virtuelle est éteinte et que ses données en mémoire sont effacées. Notre mécanisme s'appuie sur deux composants matériels qui seront présentés dans la sous-section 4.5.1 et sur une procédure logicielle. L'ensemble des étapes permettant l'arrêt d'une machine virtuelle est détaillé dans la sous-section 4.5.2.

4.5.1 Composants matériels en charge de l'arrêt d'une machine virtuelle : SVM Controller et SVM Agent

Comme dit précédemment, notre procédure d'arrêt s'appuie sur deux composants matériels, qui sont le *Shutdown Virtual Machine Controller* (SVM Controller) et le *Shutdown Virtual Machine Agent* (SVM Agent). Ceux-ci permettent d'assurer que la machine virtuelle sera bien arrêtée à la fin de la procédure. En d'autres termes, une fois la procédure démarrée, celle-ci ne peut pas être interrompue, augmentant ainsi la sécurité des machines virtuelles s'exécutant sur la plateforme.

Le composant SVM Controller est le composant maître vu par l'hyperviseur, et sert à initier la procédure d'arrêt. Le SVM Controller délègue à ses agents l'arrêt des machines virtuelles. Le composant SVM Controller se trouve dans le cluster hyperviseur alors que les SVM Agents sont répliqués dans tous les clusters réservés aux machines virtuelles.

SVM Controller

La figure 4.22 présente la vue globale du composant SVM Controller. Comme on peut le voir, ce composant est principalement composé de deux machines d'états finis, `CMD_FSM` et `CONFIG_FSM`, et de canaux. Il y a autant de canaux que de machines virtuelles pouvant être déployées simultanément. Les canaux contiennent un jeu de registre configurable donnant des informations sur la machine virtuelle associée au canal. L'automate `CONFIG_FSM` permet d'envoyer des ordres au SVM Controller, typiquement l'arrêt d'une machine virtuelle, et de configurer les registres d'un canal lors du démarrage d'une machine virtuelle. L'automate `CMD_FSM` permet au SVM Controller de mettre en place la procédure d'arrêt d'une machine virtuelle en envoyant un ordre d'arrêt aux SVM Agents présents dans les clusters de la machine virtuelle que l'on désire arrêter. Pour savoir quels SVM Agents le SVM Controller doit contacter, celui-ci utilise les informations contenues dans les registres de configuration.

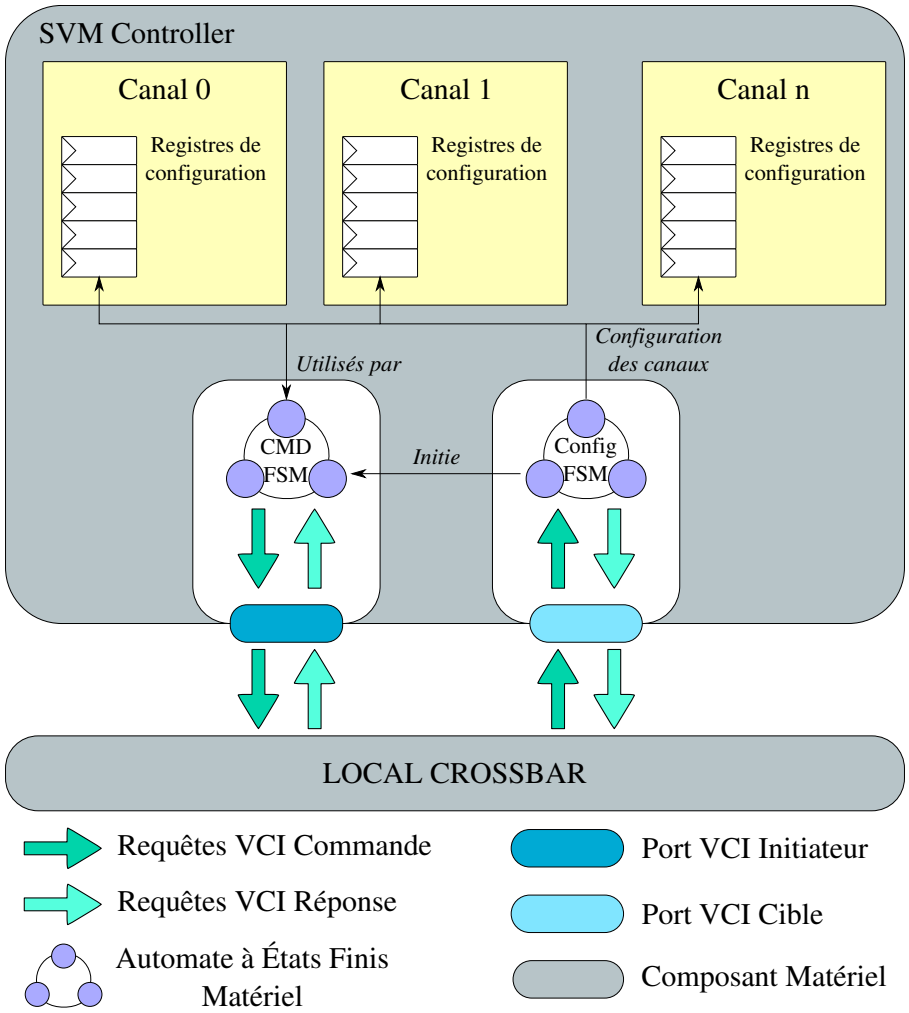

FIGURE 4.22 – Vue globale d'un SVM Controller

La table 4.4 présente les différents registres adressables du composant SVM Controller. Les registres de configuration sont séparés en deux parties, une partie de configuration et une partie de commande. L'hyperviseur et les machines virtuelles n'ont accès qu'à la partie de commande, permettant de déclencher la procédure d'arrêt. La partie de configuration n'est accessible que par les cœurs exécutant le code RHA, donc lorsque les HATs sont désactivés, au démarrage d'une machine virtuelle. Ceci permet d'assurer que la configuration du SVM Controller ne peut pas être altérée pendant la vie de la machine virtuelle.

`CTRL_VM_ID` : ce registre est répliqué dans chaque canal et contient l'identifiant de la machine virtuelle associée au canal.

`CTRL_VM_MX0` : ce registre est répliqué dans chaque canal et contient la coordonnée machine X du cluster $(0,0)$ de la machine virtuelle associée au canal.

`CTRL_VM_MY0` : ce registre est répliqué dans chaque canal et contient la coordonnée machine Y du

Nom du registre	Permission d'accès	Mode	Taille (en bits)
CTRL_VM_ID	Écriture	Configuration	8
CTRL_VM_MX0	Écriture	Configuration	4
CTRL_VM_MY0	Écriture	Configuration	4
CTRL_VM_PXL	Écriture	Configuration	4
CTRL_VM_PYL	Écriture	Configuration	4
CTRL_VM_STOP	Écriture	Configuration	8
CTRL_VM_STOP_VM	Écriture	Commande	3

TABLE 4.4 – Table des registres adressables du SVM Controller

cluster $(0, 0)$ de la machine virtuelle associée au canal.

CTRL_VM_PXL : ce registre est répliqué dans chaque canal et contient la taille en coordonnée X de la machine virtuelle associée au canal.

CTRL_VM_PYL : ce registre est répliqué dans chaque canal et contient la taille en coordonnée Y de la machine virtuelle associée au canal.

CTRL_VM_STOP : une écriture à l'adresse de ce registre permet d'agir sur le registre R_COUNT. Le registre R_COUNT est répliqué par canal et contient la valeur du nombre de réponses à recevoir pour s'assurer qu'une phase de la procédure d'arrêt est terminée. Autrement dit, il sert à synchroniser différents SVM Agents d'une même machine virtuelle, en comptant le nombre de réponses à recevoir de la part de ces SVM Agents. Lorsqu'un SVM Agent a terminé une phase de la procédure d'arrêt, celui-ci le signale au SVM Controller. Lorsque le SVM Controller a reçu tous les signalements, la procédure peut passer à la phase suivante.

CTRL_VM_STOP_VM : une écriture à l'adresse de ce registre agit sur le registre R_MODE, qui permet alors d'initier la procédure d'arrêt d'une machine virtuelle. Ce registre est répliqué par canal, et contient l'état de la machine virtuelle associée au canal. Le registre R_MODE peut prendre 5 valeurs : *VM_NONE* signifiant que le canal n'est pas associé à une machine virtuelle, *VM_RUNNING* signifiant qu'une machine virtuelle est associée au canal et que celle-ci est en cours de fonctionnement, *VM_STOP_REQUESTED* signifiant que la machine virtuelle a reçu un ordre d'arrêt, *VM_FIRST_STEP* signifiant que la machine virtuelle est dans la première phase de la procédure d'arrêt et *VM_SECOND_STEP* qui indique que la machine virtuelle est dans la deuxième phase d'arrêt.

SVM Agent

La figure 4.23 présente la vue globale d'un composant SVM Agent. Comme le montre la figure, ce composant comporte deux machines d'états finis `CMD_FSM` et `CONFIG_FSM`, un banc de registre de configuration et un fil *soft reset* relié à tous les cœurs du cluster. L'automate `CONFIG_FSM` permet de recevoir l'ordre d'arrêt d'une machine virtuelle provenant du SVM Controller, de recevoir les messages de terminaison, en provenance des cœurs, de la procédure logicielle d'arrêt, et de configurer les registres du SVM Agent lors du démarrage d'une machine virtuelle. L'automate `CMD_FSM` permet au SVM Agent de communiquer avec le SVM Controller pour l'informer de la fin d'une phase de la procédure d'arrêt et aussi de désactiver les HATs une fois la procédure finie. Le signal *soft reset* permet de brancher les cœurs du cluster vers la procédure logicielle d'arrêt des machines virtuelles.

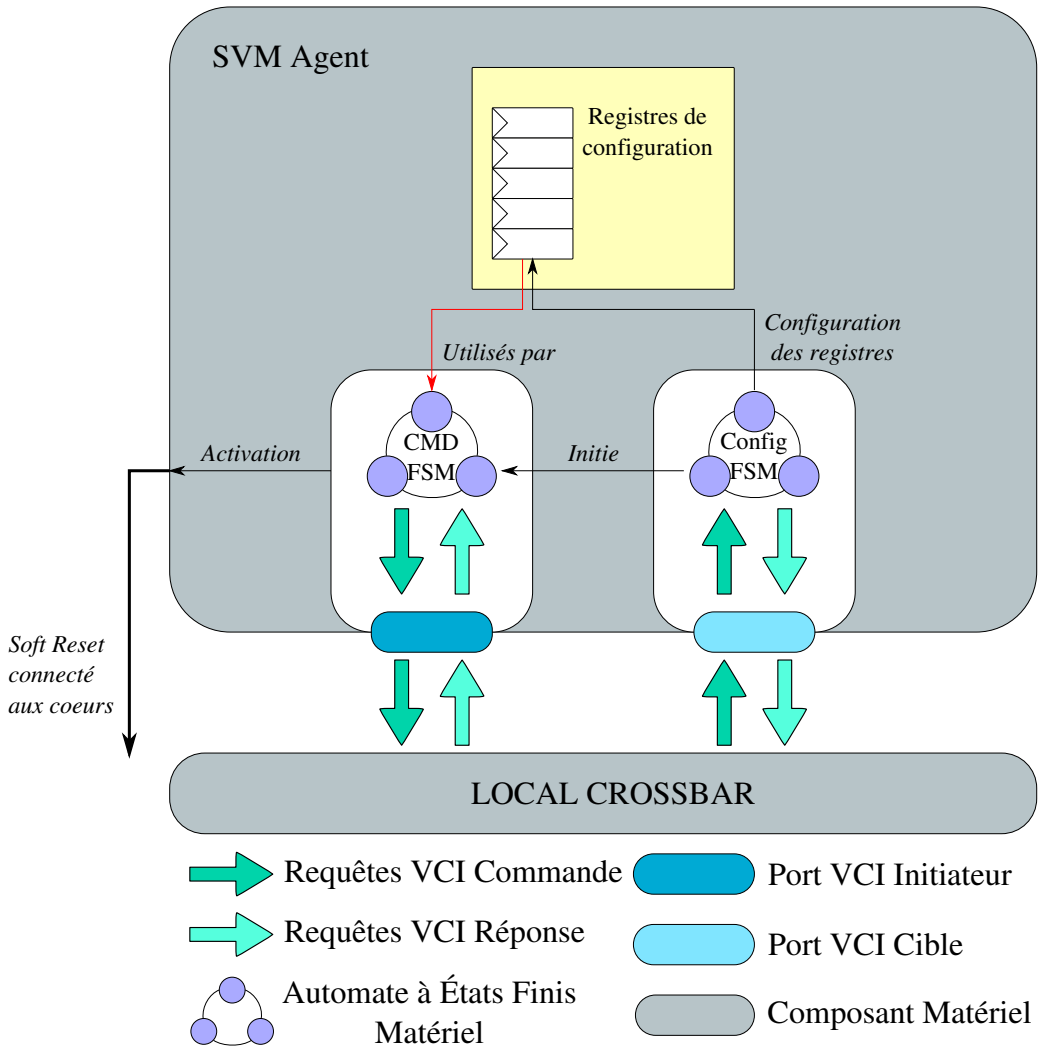

FIGURE 4.23 – Vue globale d'un SVM Agent

La table 4.5 présente les différents registres adressables du SVM Agent. Ceux-ci sont divisés en deux parties, une partie de configuration et une partie de commande. Les machines virtuelles et le

SVM Controller n'ont accès qu'à la partie de commande. La partie de configuration n'est accessible en écriture qu'au démarrage d'une machine virtuelle. Une fois que la machine virtuelle est démarrée, les registres de configuration passent en mode lecture seule. Ceci permet d'assurer que la configuration d'un SVM Agent ne peut pas être altérée pendant la vie de la machine virtuelle.

Nom du registre	Permission d'accès	Mode	Taille (en bits)
AGT_VM_STOP	Écriture	Commande	2
AGT_VM_COMPLETE	Écriture	Commande	2
AGT_VM_ID	Écriture/Lecture	Configuration	8
AGT_VM_STATUS	Lecture	Configuration	2
AGT_VM_MX0	Écriture/Lecture	Configuration	4
AGT_VM_MY0	Écriture/Lecture	Configuration	4
AGT_VM_PXL	Écriture/Lecture	Configuration	4
AGT_VM_PYL	Écriture/Lecture	Configuration	4
AGT_VM_TYPE	Écriture/Lecture	Configuration	3
AGT_VM_BDEV_ADDR	Écriture	Configuration	32
AGT_VM_DMA_ADDR	Écriture	Configuration	32
AGT_VM_HCRYPT_ADDR	Écriture	Configuration	32
AGT_VM_PROC_ADDR	Écriture	Configuration	32

TABLE 4.5 – Table des registres adressables des SVM Agents

AGT_VM_STOP : ce registre est accessible seulement par le SVM Controller grâce à une vérification faite au niveau du *SRCID*. Il permet de recevoir les ordres d'arrêt des machines virtuelles et de déclencher la procédure d'arrêt. Une écriture par le SVM Controller permet d'agir sur le registre R_STATUS, qui donne une information sur l'état d'avancement de la procédure d'arrêt. Il peut y avoir 4 valeurs pour le registre R_STATUS : *VM_NONE* signifiant qu'aucune machine virtuelle n'est actuellement en cours de fonctionnement sur le cluster, *VM_RUNNING* signifiant qu'une machine virtuelle s'exécute sur le cluster, *VM_FIRST_STEP* signifiant que la première phase de la procédure d'arrêt vient d'être enclenchée par le SVM Controller et enfin *VM_SECOND_STEP* qui indique que la procédure d'arrêt peut passer dans la deuxième phase.

AGT_VM_COMPLETE : accessible seulement par les cœurs du cluster associé au SVM Agent, il permet de synchroniser les cœurs du cluster lors de la procédure logicielle d'arrêt.

AGT_VM_ID : contient l'identifiant de la machine virtuelle s'exécutant sur le cluster.

AGT_VM_STATUS : permet aux cœurs du cluster de connaitre l'état d'avancement de la procédure d'arrêt en accédant au registre R_STATUS.

AGT_VM_MX0 : contient la coordonnée machine X du cluster $(0, 0)$ de la machine virtuelle s'exécutant

sur le cluster.

AGT_VM_MY0 : contient la coordonnée machine Y du cluster $(0,0)$ de la machine virtuelle s'exécutant sur le cluster.

AGT_VM_PXL : contient la taille en coordonnée X de la machine virtuelle s'exécutant sur le cluster.

AGT_VM_PYL : contient la taille en coordonnée Y de la machine virtuelle s'exécutant sur le cluster.

AGT_VM_TYPE : contient le type du système d'exploitation de la machine virtuelle s'exécutant sur le cluster. Ce registre peut prendre 2 valeurs : *VM_TYPE_32* dans le cas d'un système d'exploitation 32 bits ou *VM_TYPE_40* dans le cas d'un système d'exploitation 40 bits.

AGT_VM_BDEV_ADDR : contient les 32 bits de poids faible de l'adresse physique du composant HAT de l'IOC.

AGT_VM_DMA_ADDR : contient les 32 bits de poids faible de l'adresse physique du composant HAT du DMA contenu dans le cluster.

AGT_VM_HCRYPT_ADDR : contient les 32 bits de poids faible de l'adresse physique du composant HAT du HCRYPT contenu dans le cluster.

AGT_VM_PROC_ADDR : contient les 32 bits de poids faible de l'adresse physique des composants HATs des cœurs contenus dans le cluster.

4.5.2 Procédure d'arrêt d'une machine virtuelle

La procédure d'arrêt est une séquence faisant interagir les composants matériels SVM Controller et Agents avec les cœurs de la machine virtuelle. Elle est composée de deux phases, entre lesquelles une barrière de synchronisation est nécessaire. Pour la suite, il est nécessaire de distinguer trois types de cœurs : le cœur C0-VM, qui est le cœur de démarrage de la machine virtuelle ; les cœurs 0 locaux C0-L, qui sont les cœurs d'identifiant local 0 dans chaque cluster ; et les autres cœurs, que nous appellerons "normaux", ou Cx. En fonction du type de cœur, certaines étapes de la procédure ne sont pas exécutées.

La figure 4.24 représente les différentes étapes nécessaires pour l'arrêt d'une machine virtuelle. Dans cette figure, nous nous plaçons dans le scénario où une machine virtuelle déployée sur un seul cluster est arrêtée par l'utilisateur via une demande sur le terminal hyperviseur. Six entités sont en jeu : le C-Hyp, le C0-VM, un C0-L, un Cx, le SVM Controller, le SVM Agent du cluster alloué

à la machine virtuelle, ainsi qu'un deuxième SVM Agent pour mettre en avant le mécanisme de synchronisation entre SVM Agents dans le cas où il y aurait d'avantage de clusters.

Requête de l'utilisateur

Tout d'abord l'utilisateur demande à l'hyperviseur, via le terminal, l'arrêt de sa machine virtuelle. L'utilisateur doit passer en argument l'identifiant de la machine virtuelle qu'il souhaite éteindre. Avant d'enclencher la procédure d'arrêt, l'hyperviseur vérifie au préalable que la machine virtuelle ciblée est bien en cours de fonctionnement. Après cette vérification l'hyperviseur fait une requête au SVM Controller en indiquant le numéro d'instance de la machine virtuelle qu'il souhaite éteindre. Pour réaliser cela, l'hyperviseur fait une requête d'écriture dans le registre adressable `CTRL_VM_STOP_VM` avec la valeur de l'identifiant.

Appel au SVM Controller

Lorsque le SVM Controller reçoit l'écriture en provenance de l'hyperviseur, celui-ci vérifie que l'identifiant correspond à une machine virtuelle en cours de fonctionnement. Si la machine virtuelle qui doit être arrêtée n'est pas en fonctionnement, le SVM Controller renvoie une erreur à l'hyperviseur. Si la machine virtuelle est bien active, alors la procédure d'arrêt commence. Le SVM Controller place le registre `R_MODE` du canal associé à la machine virtuelle à la valeur *VM_STOP_REQUESTED*. Le SVM Controller calcule alors le nombre de composants SVM Agents qu'il doit contacter, en utilisant les registres `R_PXL` et `R_PYL` du canal de la machine virtuelle. L'automate `CMD_FSM` prend alors le relais et commence à procéder à l'envoi des commandes d'arrêt à destination des SVM Agents associés à la machine virtuelle. Pour cela, il doit calculer les coordonnées des clusters de la machine virtuelle en s'aidant des registres `R_MX0`, `R_MY0`, `R_PXL` et `R_PYL`.

Une fois toutes les commandes envoyées, le SVM Controller se met en attente des réponses des différents SVM Agents et change le statut de la machine virtuelle en *VM_FIRST_STEP* marquant ainsi le début de la première phase de la procédure d'arrêt.

Activation du signal Soft Reset

À la réception du message d'arrêt, les SVM Agents vont activer le signal *soft reset*. Ce dernier est un signal câblé entre le SVM Agent et tous les cœurs du cluster. Il permet de faire sauter les cœurs à leur adresse de `reset` (0xBFC00000 pour les processeurs MIPS). Le code présent à cette adresse appartient à la TCB de l'hyperviseur. Une fois le signal *soft-reset* activé, le SVM Agent se met en

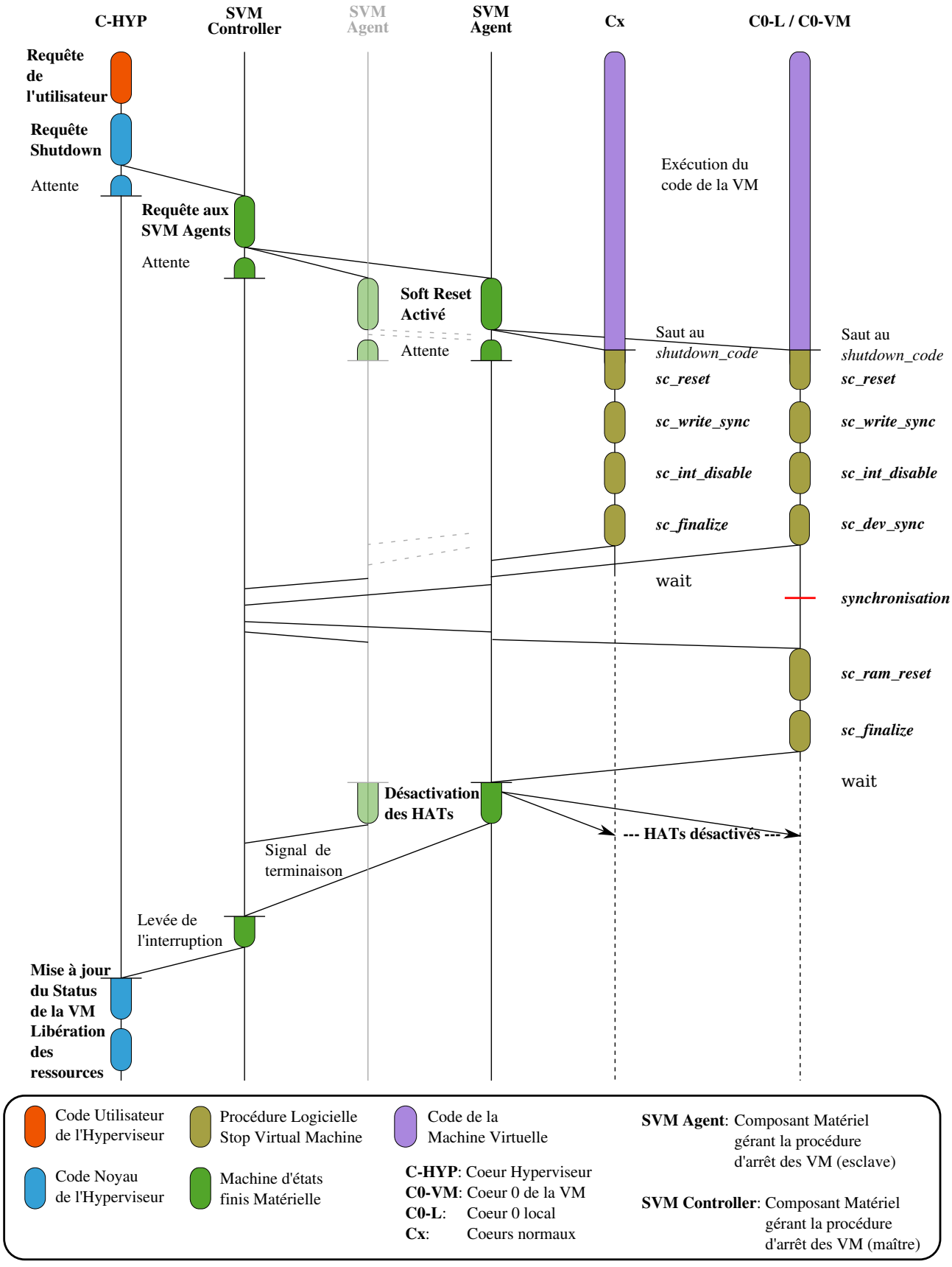

FIGURE 4.24 – Procédure d'arrêt d'une Machine Virtuelle

attente jusqu'à ce qu'il ait reçu, pour chaque cœur du cluster, un signal indiquant la fin de la première phase.

Procédure logicielle d'arrêt

Une fois les cœurs branchés au code de `reset`, ils vérifient la cause du `reset` via le registre `STATUS` du cœur. Le cœur sait que la cause est un *soft reset* car le bit éponyme du registre `STATUS` est à 1. Le cœur exécute donc le code spécifique à l'arrêt de la machine virtuelle, appelé `shutdown_code (sc)`, qui comporte six parties :

- `sc_reset` : Allocation de la pile pour et détermination de la nature de la machine virtuelle
- `sc_write_sync` : Synchronisation des écritures en cours
- `sc_int_disable` : Désactivation des interruptions
- `sc_dev_sync` : Synchronisation des requêtes en cours sur les périphériques (externes et répliqués)
- `sc_ram_reset` : Effacement de la mémoire utilisée par la machine virtuelle
- `sc_finalize` : Finalisation de la procédure logicielle

Les parties `sc_write_sync`, `sc_int_disable` et `sc_dev_sync` constituent la première phase, et la partie `sc_ram_reset` constitue la deuxième phase.

Dans le `sc_reset`, le cœur commence par consulter le SVM Agent de son cluster pour connaître la nature du système d'exploitation qu'il était en train exécuter, afin de savoir dans quel mode de fonctionnement se trouvent les HATs. Cette étape est nécessaire, car les HATs ne fournissent pas les mêmes traductions pour un système 32 ou 40 bits. Pour récupérer cette information, le cœur fait une requête de lecture dans le registre `AGT_VM_TYPE` du composant SVM Agent de son cluster. À partir de cette information, le `sc_reset` peut calculer les traductions que vont réaliser les HATs, afin de pouvoir atteindre les composants qu'il faut durant la procédure d'arrêt. Si le système était 40 bits, le cœur doit alors transposer son numéro de cluster physique, contenu dans l'identifiant du cœur, par le numéro de cluster machine. Pour calculer la coordonnée machine de son cluster, il utilise les informations contenues dans le SVM Agent, à savoir les registres `R_MX0`, `R_MY0`, `R_PXL` et `R_PYL`.

Une fois que le cœur sait dans quel mode son HAT fonctionne, il alloue une pile pour l'exécution de la procédure d'arrêt logicielle. Cette pile est allouée localement, c'est-à-dire dans son cluster, puis initialisée à 0. Cela permet d'assurer que l'espace alloué pour la pile ne contient aucune donnée provenant de la machine virtuelle anciennement exécutée. Une fois la pile allouée et initialisée, le cœur passe au code `sc_write_sync`.

Le code `sc_write_sync` fait une synchronisation des écritures en cours, pour s'assurer que tous les accès mémoire commencés par la machine virtuelle sont terminés. Le code `sc_int_disable` désactive les interruptions en entrée de l'XICU, sauf une qui sera utilisée pour le réveil des cœurs.

Les cœurs doivent ensuite s'assurer que les périphériques externes alloués à la machine virtuelle ont fini toutes les requêtes en cours. Par exemple, la machine virtuelle peut avoir été éteinte alors que celle-ci faisait un accès au disque ; il faut alors s'assurer que l'accès au disque soit terminé avant de libérer la zone réservée à la machine virtuelle. De même, ils doivent s'assurer que les périphériques répliqués ne sont pas en cours de fonctionnement, pour la même raison que les périphériques externes. Ces vérifications sont faites par le code `sc_dev_sync`, mais pas par tous les cœurs : les C0-L synchronisent les périphériques répliqués, tandis que le C0-VM synchronise les périphériques externes. À la fin du code `sc_dev_sync`, les cœurs envoient un signal de fin de phase au SVM Agent de leur cluster, et attendent la mise à jour du statut du SVM Agent avant de passer à la deuxième phase.

Les cœurs normaux ne réalisent eux aucune opération de synchronisation et passent directement au code `sc_finalize`. Ce code est composé de quatre étapes :

- Réinitialiser puis désallouer la pile allouée dans le code `sc_init`.
- Invalider le contenu du cache de premier niveau. Cette invalidation doit être faite pour éviter des problèmes de cohérence mémoire.
- Signaler la fin de la procédure au SVM Agent de leur cluster
- Se mettre dans l'état basse consommation en exécutant l'instruction `wait`.

Lorsqu'un SVM Agent a reçu les signaux de fin de phase de tous les cœurs de son cluster, il envoie à son tour un signal de fin de phase au SVM Controller. Lorsque le SVM Controller a reçu un signal de fin de phase de la part de tous les SVM agents de la machine virtuelle, il change son statut en *VM_SECOND_STEP*, et envoie un message à tous les SVM Agents pour leur signaler le début de la deuxième phase. À la réception de ce message, ces derniers mettent à leur tour leur statut à jour en *VM_SECOND_STEP*. Cela débloque ainsi les cœurs C0-L qui faisaient de la scrutation sur ce statut.

Cette synchronisation est donc réalisée sous la forme d'une barrière hiérarchique, de manière à avoir un mécanisme distribué, localisé (tous les accès aux SVM Agents sont locaux), uniforme (tous les SVM Agents ont le même comportement) et avoir un point de centralisation, le SVM Controller, moins contraint.

Une alternative à cette solution aurait été d'avoir une barrière logicielle. Néanmoins, une résolution logicielle de ce problème de synchronisation est très complexe, et nous ne sommes pas sûrs de pouvoir réaliser une telle solution sans support matériel. En effet, l'initialisation de la barrière doit être faite une seule fois et à une adresse connue de tous les cœurs. Ces deux points sont problématiques :

- L'initialisation d'une barrière se fait généralement dans une phase séquentielle, ou alors à l'aide d'une valeur booléenne initialisée dans le binaire ; or ni l'une ni l'autre de ces options n'est possible ici.
- Les différents cœurs des clusters de la machine virtuelle peuvent difficilement communiquer entre eux pour partager l'adresse de la structure de la barrière, car les piles sont allouées localement et dynamiquement par cluster, et il n'y a pas de mémoire de données globales.

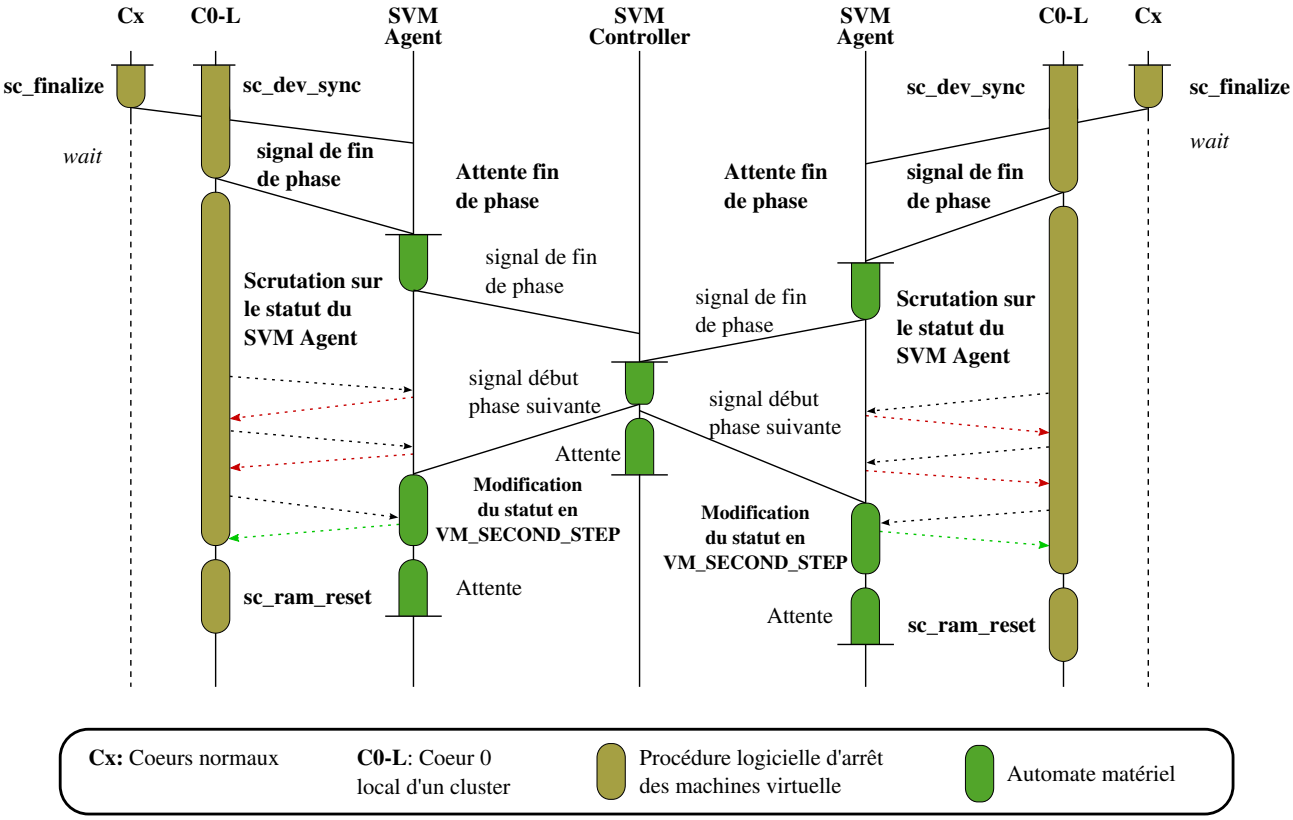

FIGURE 4.25 – Mécanisme de synchronisation des SVM Agents

Ce sont pour ces raisons que nous avons choisi un mécanisme de barrière matérielle.

La figure 4.25 illustre le mécanisme de synchronisation mis en œuvre, qui fait intervenir 7 entités : 2 cœurs normaux (Cx), 2 cœurs 0 locaux (C0-L), 2 SVM Agents et le SVM Controller.

La deuxième phase logicielle exécutée par les cœurs C0-L, correspondant au code `sc_ram_reset`, consiste à effacer les bancs mémoire. Cette phase occupe la majeure partie du temps passé dans la procédure d'arrêt. Pour éviter que le temps augmente avec le nombre de clusters alloués à la machine virtuelle, nous avons fait en sorte que chaque cœur C0-L efface le banc mémoire contenu dans son cluster. L'effacement de la mémoire allouée à la machine virtuelle a pour but d'assurer la confidentialité des données de la machine virtuelle.

L'effacement de la mémoire est effectué à l'aide du DMA présent dans le cluster, permettant ainsi d'accélérer le traitement de cette phase. Contrairement aux périphériques externes, le DMA n'écrit pas directement en mémoire mais dans le cache L2. Une fois les écritures terminées, il faut donc synchroniser les caches L2 pour assurer que la mémoire contient bien les dernières valeurs écrites, c'est-à-dire rendre effectif l'effacement de la mémoire.

Il est important de noter qu'il ne faut pas effacer l'intégralité de la mémoire. En effet, il ne faut pas effacer la mémoire allouée pour la pile, sans quoi des informations cruciales seraient perdues : par exemple, l'adresse de retour de la procédure d'arrêt. Ensuite, les cœurs C0-L exécutent le code `sc_finalize` défini précédemment. Le signalement de fin de phase contenu dans le `sc_finalize` débloque le SVM Agent pour la suite de la procédure.

Désactivation des HATs

Lorsque le SVM Agent a reçu le signalement de fin de phase du cœur C0-L de son cluster, il désactive alors tous les HATs de son cluster – aussi bien ceux des cœurs, des DMAs, que des périphériques alloués à la machine virtuelle. Seul le SVM Agent du cluster $(0, 0)$ de la machine virtuelle désactive les HATs des périphériques externes. Pour pouvoir désactiver les HATs, le SVM Agent utilise les informations contenues dans ses registres de configuration et effectue une requête vers le port de configuration des HATs. Une fois tous les HATs désactivés, le SVM Agent envoie un message de terminaison au SVM Controller indiquant que la procédure d'arrêt du cluster est terminée. Ainsi, les HATs ne sont désactivés qu'après l'effacement de toutes les mémoires et la libération de tous les périphériques. La garantie que les HATs ne soient désactivés que par le SVM Agent est assurée par une contrainte sur le *SRCID*.

Mise à jour des structures de l'hyperviseur

La réception de tous les messages de fin vers le SVM Controller constitue une seconde barrière matérielle, qui synchronise la fin de la procédure entre tous les clusters. Ainsi, lorsque le SVM Controller a reçu tous les messages de terminaison des SVM Agents associés à la machine virtuelle, celui-ci lève une interruption à l'intention de l'hyperviseur lui signifiant que la machine virtuelle est éteinte. L'hyperviseur met à jour le statut de la machine virtuelle dans ses structures de données et libère les clusters anciennement utilisés par la machine virtuelle. Ceux-ci peuvent ainsi être utilisés pour démarrer une autre machine virtuelle.

4.5.3 Conclusion

Dans cette section, nous avons présenté notre mécanisme d'arrêt des machines virtuelles. Ce mécanisme est une procédure logicielle assistée par le matériel, qui peut être initiée par l'hyperviseur ou par les machines virtuelles elles-mêmes. Deux composants sont en charge d'assurer le bon fonctionnement de la procédure d'arrêt : le *Shutdown Virtual Machine Controller* et les *Shutdown Virtual Machine Agents*. Ces composants enclenchent la procédure d'arrêt et assurent qu'à la fin de celle-ci, la machine

virtuelle est bien arrêtée et que la mémoire allouée à la machine virtuelle est effacée. Ces composants sont aussi les seuls à pouvoir désactiver les HATs, étape nécessaire avant le re-déploiement d'une nouvelle machine virtuelle.

4.6 Chiffrement du disque dur de la machine virtuelle

Comme nous l'avons vu dans la sous-section 4.3.2, chaque canal du MULTI_IOC contient un disque dur (ou une partition). Ce dernier est associé à une machine virtuelle et contient l'image du noyau, le système de fichier et le `bootloader` du système d'exploitation associé. Tous les disques sont chiffrés avec une clé spécifique à un utilisateur, qui peut être retrouvée à partir d'un mot de passe utilisateur. Les mécanismes de chiffrement du disque et de déchiffrement sont présentés dans les sous-sections 4.6.3 et 4.6.4. Tous les principaux calculs cryptographiques sont réalisés à l'aide d'un crypto-processeur sécurisé intégré dans l'architecture. Celui-ci est nommé HCrypt et est présenté dans la sous-section 4.6.1 ci-dessous.

4.6.1 Coprocesseur cryptographique : HCrypt

Le cryptoprocesseur HCrypt a été développé au cours de la thèse de Gaspar Lubos [92] au sein du laboratoire Hubert Curien. HCrypt utilise des clés symétriques, le chiffrement et le déchiffrement sont donc effectués à l'aide de la même clé. Ce genre d'algorithme est généralement plus efficace que les algorithmes à clés publiques qui utilisent des opérations bien plus coûteuses en terme de temps de calcul. Les opérations utilisées pour le chiffrement symétrique sont plus simples, et sont généralement des opérations logiques (xor, décalage) et arithmétiques simples (additions), contrairement aux opérations requises pour le chiffrement asymétrique qui utilise des multiplications sur des grands entiers.

Une primitive de base des algorithmes symétriques est le bloc de chiffrement, qui traite des blocs d'information de taille fixe. Un bloc de chiffrement peut uniquement chiffrer des messages d'une taille égale à celle du bloc. Pour chiffrer des informations de plus grande taille, un chainage entre les blocs est utilisé, les plus courants étant CFB, CBC, OFB, et *Counter* [93]. Dans les sous-sections suivantes, nous décrirons le mode *Counter*.

Présentation de HCrypt

Le HCrypt fournit trois services de sécurité : une gestion des clés sécurisée, la confidentialité et l'intégrité des données.

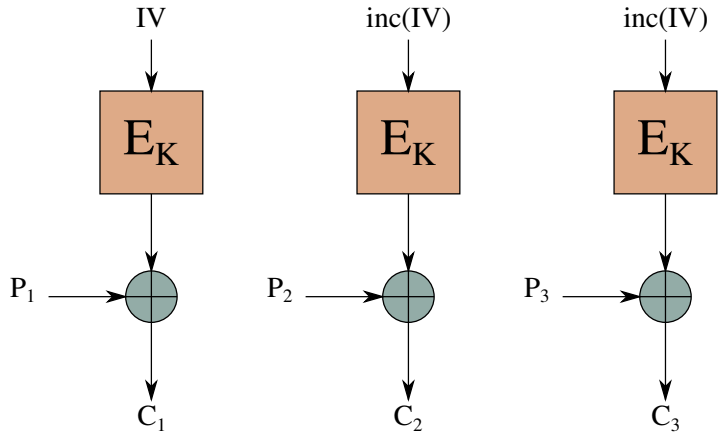

FIGURE 4.26 – Structure de l'algorithme *Counter*

Le service de gestion des clés fournit par HCrypt est sécurisé par la présence d'une clé maître K_M qui est chargée pendant la phase d'initialisation du crypto-processeur. Toutes les clés de session K_S sont générées en utilisant un *True Random Number Generator* (TRNG). Si une clé de session est demandée, par exemple par l'utilisateur, celle-ci est d'abord chiffrée en utilisant le cœur de chiffrement *Advance Encryption Standard* (AES) du crypto-processeur et la clé maître K_M puis celle-ci est donnée en sortie du crypto-processeur. Lorsqu'une clé de session est chargée au sein du HCrypt, celle-ci est au préalable déchiffrée à l'aide de l'AES et de la clé maître avant d'être stockée dans un registre interne. Ce mécanisme de chiffrement des clés de session lorsque celles-ci sortent du HCrypt garantit que seul le HCrypt peut déchiffrer des données en les utilisant.

Le chiffrement et le déchiffrement des données dans HCrypt utilisent l'AES avec un chainage de type *Counter*. Ce chainage a pour avantage que le chiffrement ou déchiffrement d'un bloc est indépendant des autres blocs. La figure 4.26 illustre la structure du mode *Counter*. P est le message en clair alors que C est le message chiffré. Le vecteur d'initialisation IV est composé de la concaténation d'un *nonce*, qui est un nombre arbitraire unique pour chaque message, et d'un compteur cpt tel que $IV = nonce||cpt$. La fonction $inc(IV)$ incrémente simplement la valeur du compteur. L'opération de déchiffrement est exactement la même que pour le chiffrement, le seul changement réside dans le fait que le message d'entrée est C et non P.

En ce qui concerne le chiffrement du disque mis en place dans HCrypt, nous utilisons aussi le mode *Counter* avec $IV = lba||cpt$, où lba est le numéro logique de secteur. Cela permet d'assurer que pour chaque secteur du disque l'IV est différent, tout en ne requiérant pas d'espace supplémentaire pour stocker de méta-données. D'autres algorithmes, tels que XTS [94, 95] ou XCB [96, 97], ont été étudiés pour une éventuelle implémentation au sein du HCrypt, mais aucune suite n'a été donnée à ces études.

Un service permettant d'assurer l'intégrité des données est aussi présent au sein du HCrypt sous la forme d'un algorithme *Message Authentication Code* [98] (MAC). Ces algorithmes reçoivent un mes-

sage M en entrée et produisent une étiquette d'authentification T en sortie. Le message original change alors en $M||T$. Pour vérifier l'intégrité d'un tel message, l'algorithme MAC est exécuté une nouvelle fois sur M et produit T'. Puis une comparaison est faite entre les deux étiquettes T et T' : si celles-ci sont différentes, alors le message est corrompu. Dans HCrypt, nous utilisons PMAC1 [99] pour le service d'intégrité des données.

HCrypt implémente aussi la fonction de hachage SHA3 [100] pour assurer l'intégrité de larges fichiers. La différence est que dans le cas d'un MAC, calculer le résumé (hash) du message requiert une clé, alors que dans une fonction de hachage classique, le hash dépend uniquement du message, donc un utilisateur qui peut modifier le message et le hash peut le faire de manière transparente.

L'architecture générale du HCrypt est présentée dans la figure 4.27. Celle-ci est composée de deux parties séparées, une réservée pour la gestion des clés et l'autre pour le traitement des données.

– La partie dédiée à la gestion des clés est composée d'un cœur de chiffrement et de déchiffrement AES, de deux registres permettant de stocker la clé maître et la clé de session, et d'un TRNG. Cette partie est en charge de la génération des clés en utilisant le TRNG ainsi que du chiffrement

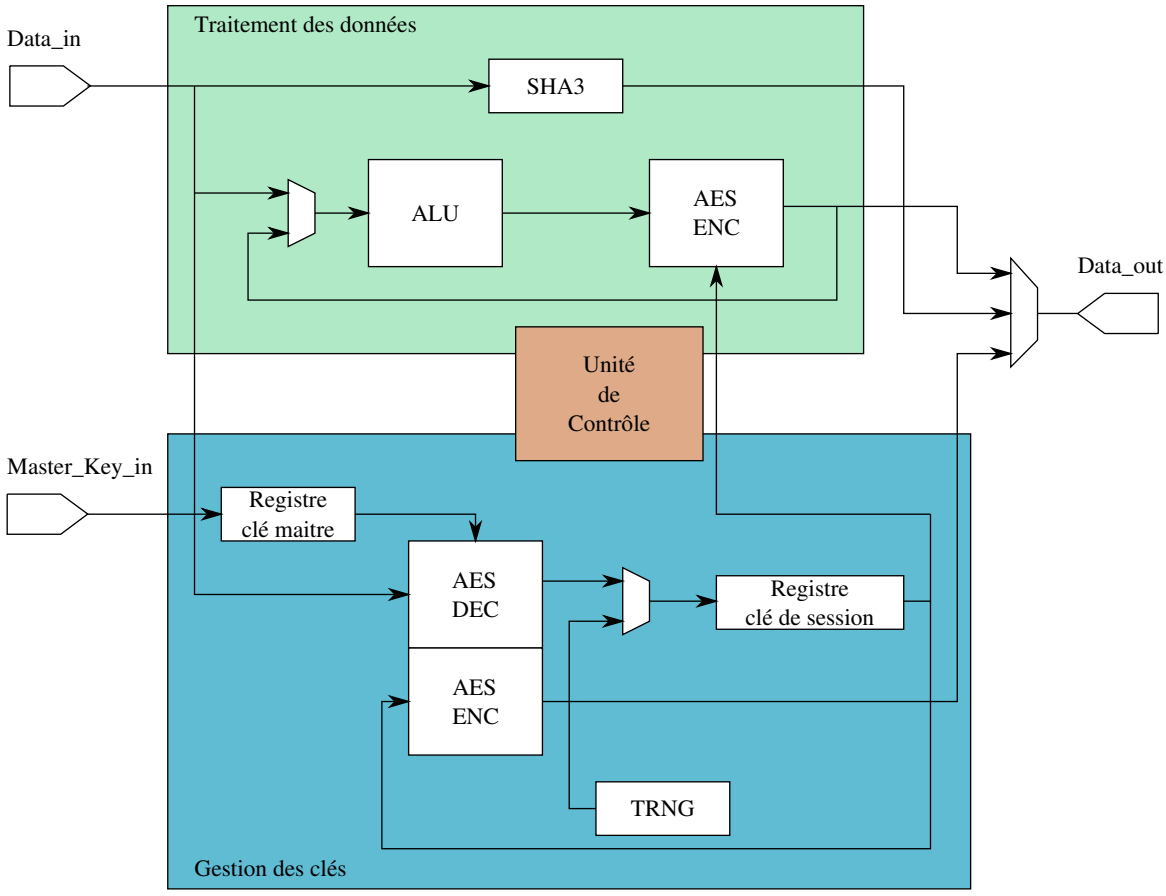

FIGURE 4.27 – Architecture interne du crypto-processeur HCrypt

et du déchiffrement des clés de session à l'aide de l'AES et de la clé maître.
– La partie dédiée au traitement de données exécute le mode *Counter* ou le PMAC. Elle possède un cœur de chiffrement AES, un cœur SHA3 et une unité arithmétique et logique pour réaliser les opérations arithmétiques nécessaires.

Intégration de HCrypt dans Tsunamy

Le cryptoprocesseur HCrypt a été intégré au sein de l'architecture Tsunamy par Marco Jankovic, stagiaire au Laboratoire d'Informatique de Paris 6, et Cuautehmoc Mancillas, post-doctorant au laboratoire Hubert Curien. Pour cela, nous l'avons couplé avec un composant DMA nommé MWMR-DMA [101], développé par Alain Greiner. Ce composant permet au HCrypt de communiquer avec les différents éléments de l'architecture via le standard de communication VCI utilisé dans l'architecture Tsunamy. La figure 4.28 illustre comment les composants HCrypt et MWMR-DMA sont interconnectés.

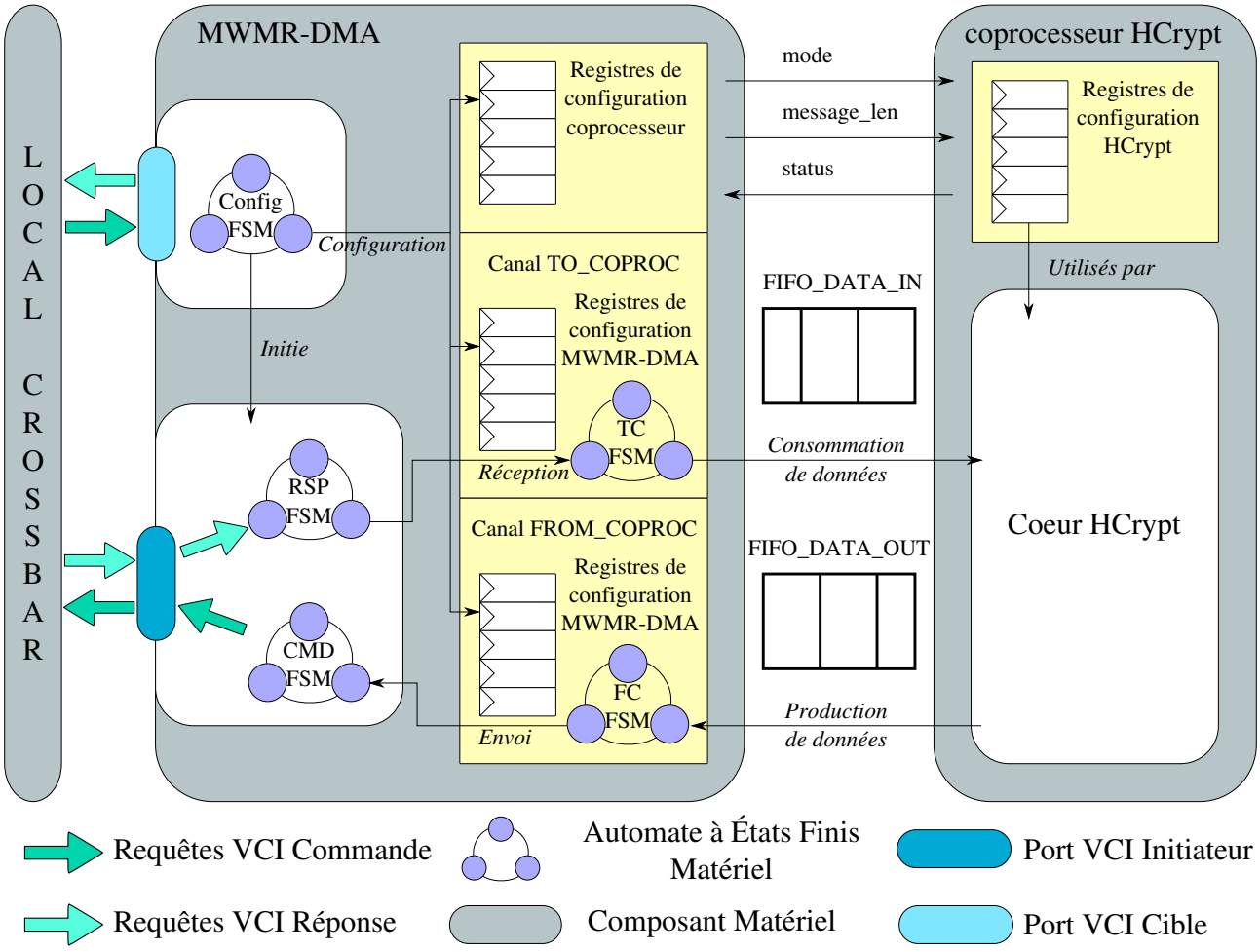

FIGURE 4.28 – Communication entre le MWMR-DMA et le cryptoprocesseur HCrypt

Le composant MWMR-DMA permet d'interconnecter, de façon générique, des coprocesseurs tels que le cryptoprocesseur HCrypt. Il est composé de cinq compartiments différents : un compartiment `CONFIG` réservé à la configuration des différents canaux du composant, un compartiment `CMD/RSP` en charge des transactions vers la mémoire, un compartiment `COPROC` permettant de configurer le coprocesseur associé au MWMR-DMA, un compartiment `TO_COPROC` en charge d'envoyer des données vers le coprocesseur et enfin un compartiment `FROM_COPROC` en charge de recevoir les données produites par le coprocesseur.

La partie `CONFIG` du MWMR-DMA sert à configurer le coprocesseur, ainsi que les canaux `TO_COPROC` et `FROM_COPROC`. Pour la configuration du coprocesseur, les commandes passées au MWMR-DMA transitent par le port de configuration `p_config`. Ce port contient un nombre de signaux 32 bits égal au nombre de registres configurables du coprocesseur. La configuration du canal `TO_COPROC` consiste principalement à donner l'adresse mémoire où se situent les données qui vont être traitées par le coprocesseur, ainsi que la taille (en octets) de l'opération que va effectuer le coprocesseur. Cette taille correspond au nombre d'octets que le MWMR-DMA va fournir au coprocesseur pour que ce dernier puisse effectuer l'opération demandée. La configuration du canal `FROM_COPROC` est similaire sauf que l'adresse désigne la zone mémoire dans laquelle les données produites par le coprocesseur vont être écrites.

Le port `p_config` comporte, dans notre cas, deux signaux 32 bits : *message_len* et *mode*. Le signal *message_len* contient la taille du message que va traiter le HCrypt, tandis que le signal *mode* contient le type d'opération que va réaliser le HCrypt. L'ensemble des valeurs possibles de ce signal est fourni dans la table 4.6.

mode (32 bits)	rôle
0 *LOAD_MASTER_KEY*	Permet de charger une clé maître
1 *GEN_SESSION_KEY*	Permet de charger une clé de session
2 *ENCRYPT_SESSION_KEY*	Permet de chiffrer une clé de session
3 *LOAD_SESSION_KEY*	Permet de charger une clé de session chiffrée
4 *PMAC*	Permet d'effectuer une opération PMAC
5 *COUNTERMODE_ENC*	Permet de chiffrer des données en mode *Counter*
6 *COUNTERMODE_DEC*	Permet de déchiffrer des données en mode *Counter*
7 *DISK_ENCRYPTION*	Permet de chiffrer un bloc du disque dur
8 *SHA3*	Permet d'effectuer une opération SHA3

TABLE 4.6 – Valeurs possibles pour le signal *mode* du port `p_config` du HCrypt

Le port `status` contient l'état actuel du HCrypt, par exemple si celui-ci est disponible, si le HCrypt a fini la tâche qu'il traitait ou s'il a rencontré une erreur pendant le traitement de l'opération. L'ensemble des valeurs possibles pour ce port sont présentées dans la table 4.7.

Valeur	Fonctionnement
IDLE	Le HCrypt est disponible
EXEC	Le HCrypt est occupé
SUCCESS	Le HCrypt a terminé la tâche en cours
ERROR	Le HCrypt a rencontré une erreur lors de l'exécution de la dernière tâche

TABLE 4.7 – Valeurs possibles pour le port `status` du HCrypt

Le port `fifo_data_in` fournit des données au HCrypt en provenance du composant MWMR-DMA. Ce port utilise le protocole *First In First Out* (FIFO) et permet d'alimenter en données le cryptoprocesseur. La largeur de ce port est de 32 bits, il faut donc 4 cycles d'horloge pour transférer 128 bits de données, soit la taille du chemin de données du cryptoprocesseur HCrypt.

Le port `fifo_data_out` fournit des données au MWMR-DMA en provenance du HCrypt. Ce port utilise également le protocole FIFO. La largeur de ce port est aussi de 32 bits, il faut donc également 4 cycles d'horloge pour recevoir les 128 bits de données produits par le HCrypt.

Le cryptoprocesseur HCrypt ayant été réalisé à la fois en SystemC et en VHDL, des informations de timing précises sont disponibles. La table 4.8 donne le temps nécessaire au HCrypt pour pouvoir réaliser les différentes opérations disponibles au sein de celui-ci. Cette table donne aussi les tailles des données consommées et produites pour chaque opération.

Opérations	Données consommées	Latence	Données produites
LOAD_MASTER_KEY	128 bits	1 cycle	0 bit
GEN_SESSION_KEY	0 bit	2 cycles	128 bits
ENCRYPT_SESSION_KEY	0 bit	11 cycles	128 bits
LOAD_SESSION_KEY	128 bits	22 cycles	0 bit
PMAC	128 bits / 11 cycles	$11 \times nBlks + 11$ cycles	128 bits
COUNTERMODE_ENC	128 bits / 11 cycles	11 cycles par bloc	128 bits / 11 cycles
COUNTERMODE_DEC	128 bits / 11 cycles	11 cycles par bloc	128 bits / 11 cycles
DISK_ENCRYPTION	128 bits / 11 cycles	11 cycles par bloc	128 bits / 11 cycles
SHA3	1152 bits / 16 cycles	$16 \times kBlks$ cycles	128 bits / 11 cycles

TABLE 4.8 – Table des temps nécessaires au HCrypt pour les différentes opérations. $nBlks = (Message_{len} * 8/128)$ et $kBlks = (Message_{len} * 8/1152)$ avec $Message_{len}$ en octet

4.6.2 Password-Based Key Derivation Function 2 (PBKDF-2)

PBKDF-2 [102, 103] est une fonction de dérivation de clé appartenant aux normes *RSA Laboratories Public-Key Cryptography Standards* (PKCS), elle remplace la fonction PBKDF-1 qui pouvait seulement produire des clés d'une longueur maximale de 160 bits.

PBDKF-2 applique une fonction pseudo-aléatoire, telle qu'une fonction de hachage ou un $HMAC$, à un mot de passe avec un sel cryptographique et réitère le processus de nombreuses fois pour produire une clé dérivée. Celle-ci peut être ensuite utilisée comme clé de chiffrement. Le nombre d'itération permet d'augmenter le temps de calcul et donc la résistance du mot de passe aux attaques en force brute. Le rajout du sel permet lui d'éviter l'utilisation d'attaques à base de dictionnaires comme les *rainbow tables* [104, 105] et donc que le hachage des mots de passe ne soit précalculé. La fonction PBKDF-2 génère une clé dérivée CD, qui a pour formule :

- $CD = PBKDF2(PRF, mdp, sel, i, CD_{len})$

Dans cette formule :

- PRF est la fonction pseudo-aléatoire utilisée à chaque itération, par exemple $HMAC$.
- mdp est le mot de passe sur lequel on applique la fonction PBKDF-2.
- sel est le sel utilisé pour la fonction cryptographique.
- i est le nombre d'itérations.
- CD_{len} est la taille en bits de la clé dérivée désirée.

La clé dérivée est composée de plusieurs blocs, ayant chacun la taille produite par la fonction PRF, que l'on concatène jusqu'à obtenir la taille de clé CD_{len} désirée :

- $CD = F(PRF, mdp, sel, i, 1)||F(PRF, mdp, sel, i, 2)||...||F(PRF, mdp, sel, i, j_{max})$

La fonction F est obtenue en réalisant le XOR de i appels à la fonction PRF. Le premier appel à cette fonction utilise comme paramètres le mot de passe ainsi que le sel concaténé avec le numéro de bloc j. Les appels suivants utilisent le mot de passe et le résultat de l'appel précédent, créant ainsi un processus itératif. On obtient donc la formule suivante :

- $F(PRF, mdp, sel, i, j) = O_{prf_1} \oplus O_{prf_2} \oplus O_{prf_3} \oplus ... \oplus O_{prf_i}$
- $O_{prf_1} = PRF(mdp, sel||j)$
- $O_{prf_2} = PRF(mdp, O_{prf_1})$
- $O_{prf_3} = PRF(mdp, O_{prf_2})$
- $O_{prf_i} = PRF(mdp, O_{prf_{i-1}})$

4.6.3 Procédure du chiffrement du disque

Dans notre architecture, les images disques, contenant le noyau et le système de fichier, sont chiffrées à l'aide d'une clé utilisateur. Cette clé utilisateur dérive d'un mot de passe. Au cours de la vie de la machine virtuelle, cette clé est stockée dans la mémoire de la machine virtuelle. Le noyau nécessite d'être déchiffré par le `bootloader` au démarrage de la machine virtuelle, et le système de fichier est déchiffré par le noyau à chaque fois qu'un nouveau bloc est lu depuis le disque dur. De

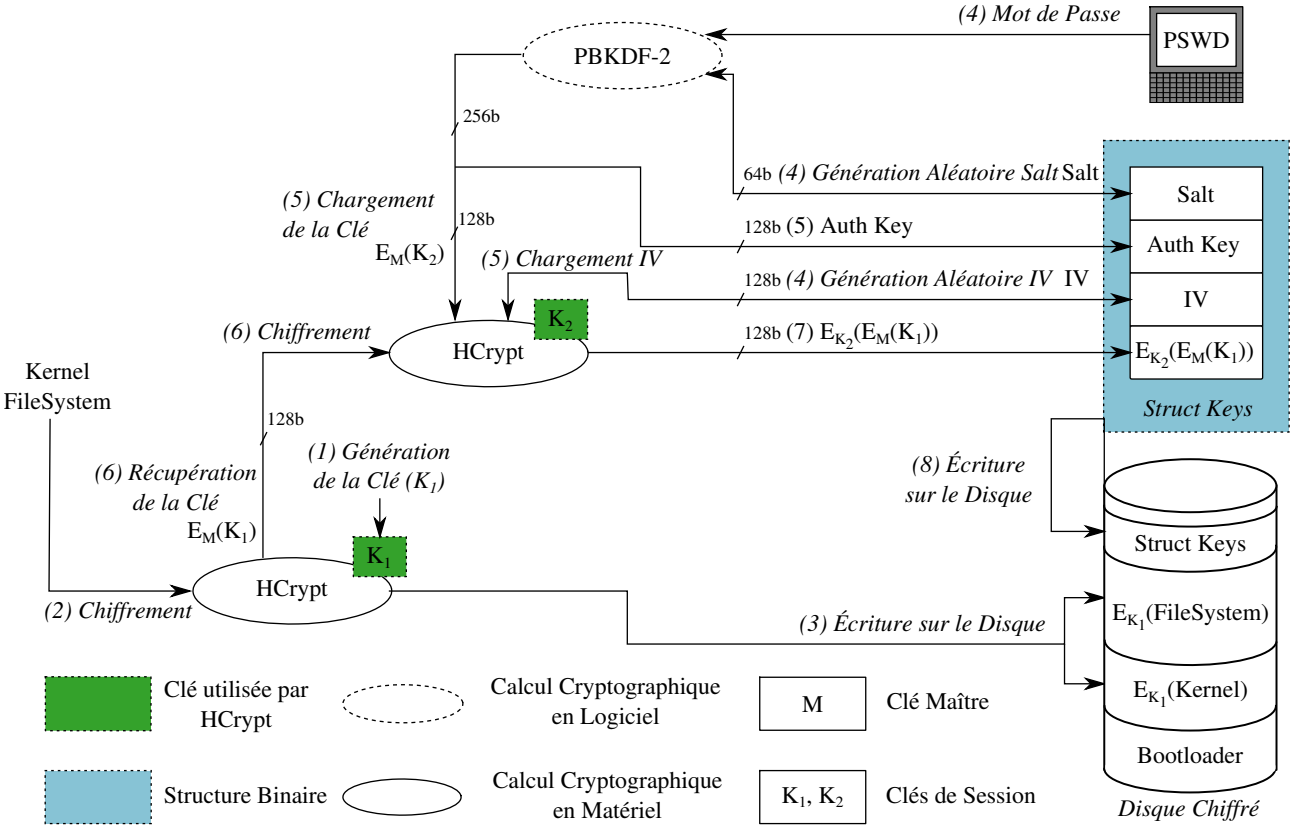

FIGURE 4.29 – Procédure de création du disque dur chiffré

façon similaire, à chaque écriture d'un nouveau bloc sur le disque dur, celui-ci est au préalable chiffré à l'aide de la clé.

La figure 4.29 illustre la procédure nécessaire pour créer un disque dur chiffré. La première étape consiste à chiffrer le noyau et le système de fichier. Pour cela, le HCrypt doit générer une clé K_1 : cette clé est la clé utilisateur utilisée pour le chiffrement des différents éléments de l'image disque. Une fois chiffré, le noyau et le système de fichier sont écrits sur le disque sous la forme $E_{K_1}(Kernel)$ et $E_{K_1}(FileSystem)$ (étapes 1, 2, 3).

La seconde et dernière étape consiste à créer une structure binaire contenant tous les éléments permettant de retrouver la clé cryptographique K_1. Cette structure sera utilisée par le `bootloader` lors de son exécution pour pouvoir déchiffrer le noyau. Pour cela, deux valeurs sont générées aléatoirement : un Vecteur d'Initialisation (IV) et un sel cryptographique (*Salt*). Le *Salt* est combiné avec le mot de passe utilisateur dans la fonction cryptographique logicielle PBKDF-2, qui permet d'avoir une clé 256 bits en sortie (étape 4).

Cette clé est alors séparée en deux parties : les 128 bits de poids fort sont utilisés en tant que Clé d'Authentification (*Auth Key*) alors que les 128 bits de poids faible sont utilisés comme clé de session $E_M(K_2)$ pour chiffrer la clé K_1. L'IV et la clé $E_M(K_2)$ sont ensuite chargés dans le HCrypt, ainsi la clé

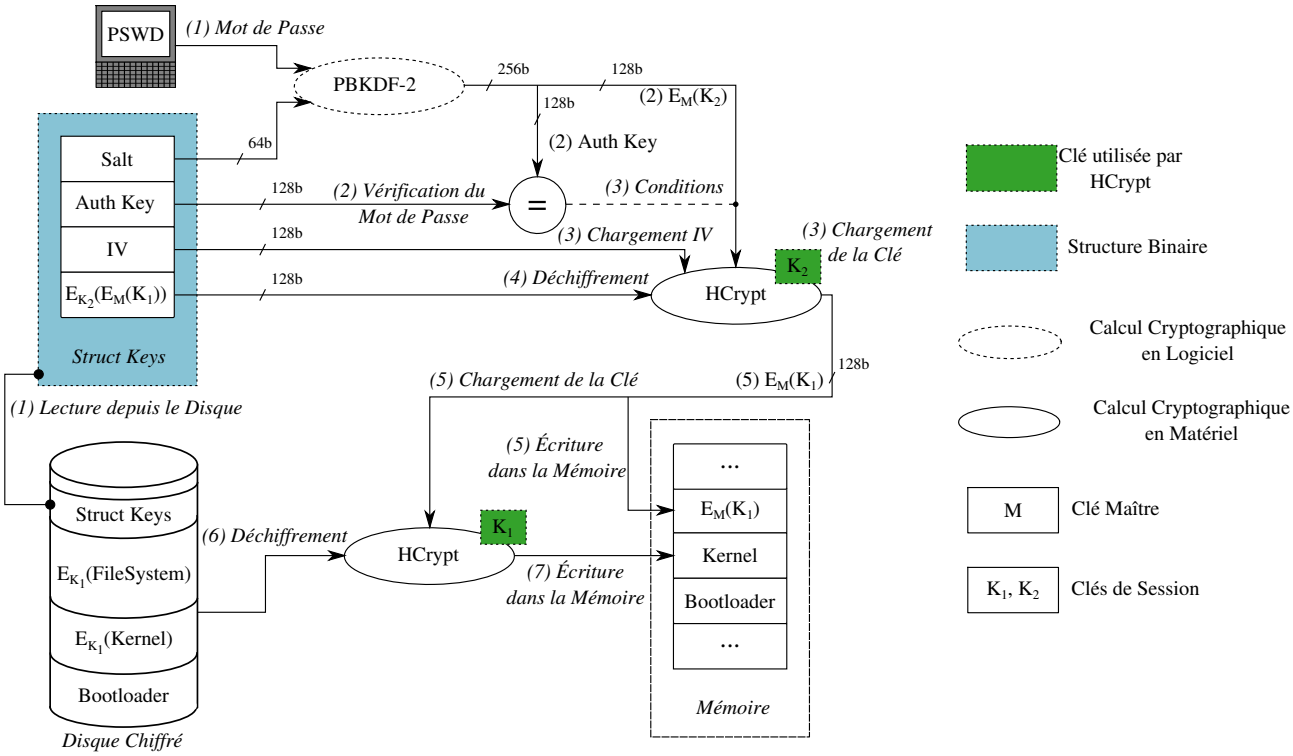

FIGURE 4.30 – Procédure de déchiffrement du disque dur chiffré

de session $E_M(K_1)$ peut être chiffrée (étapes 5, 6, 7). Les 128 bits de poids faible sont appelés $E_M(K_2)$ car lorsqu'une clé de session est chargée dans le HCrypt, celui-ci s'attend à recevoir une clé chiffrée qu'il déchiffre en interne à l'aide de sa clé maître.

Le HCrypt donne en sortie la clé de session chiffrée $K_1 : E_{K_2}(E_M(K_1))$. Cette clé chiffrée est stockée dans une structure binaire avec l'*IV*, la *Auth Key* et le *Salt*. Cette structure binaire, nommée *Struct Keys*, est finalement enregistrée dans l'image disque et sera utilisée par le `bootloader` pour permettre le déchiffrement du disque (étapes 7, 8).

4.6.4 Procédure du déchiffrement du disque

La figure 4.30 présente la procédure nécessaire pour le déchiffrement du disque dur chiffré. Cette procédure a lieu au démarrage de la machine virtuelle et est exécutée par le `bootloader` du système d'exploitation.

La structure *Struct Keys* contenue dans le disque dur est lue une fois que le mot de passe utilisateur est entré. Le `bootloader` exécute alors la fonction PBKDF-2 avec le *Salt* contenu dans la structure et le mot de passe (étape 1).

Les 128 bits de poids fort sont alors comparés à la clé *Auth Key* de la structure *Struct Keys* pour vérifier la validité du mot de passe. Si le mot de passe est faux, alors le `bootloader` s'arrête. Sinon $E_M(K_2)$ est alors chargée dans le HCrypt comme clé de session, ainsi que l'IV, permettant ainsi de déchiffrer $E_{K_2}(E_M(K_1))$ (étapes 2, 3, 4).

Une fois déchiffrée, la clé $E_M(K_1)$ est chargée dans le HCrypt et stockée en mémoire. Ceci permet au système d'exploitation d'être capable de déchiffrer d'autres données du disque dur – par exemple des applications ou des images – sans devoir demander le mot de passe utilisateur à chaque fois (étape 5).

Finalement, le `bootloader` déchiffre l'intégralité du noyau et l'enregistre en mémoire, après quoi il est capable d'exécuter le démarrage du système d'exploitation (étapes 6, 7). La clé utilisateur $E_M(K_1)$ est déchargée du HCrypt après chaque opération.

4.6.5 Conclusion

Dans cette section, nous avons présenté notre solution concernant la confidentialité des disques des machines virtuelles. Pour cela, nous proposons une procédure permettant de chiffrer puis de déchiffrer l'image disque des machines virtuelles, i.e. le noyau du système d'exploitation et son système de fichiers. Pour les opérations de chiffrement et de déchiffrement, nous utilisons un crypto-processeur sécurisé, nommé HCrypt, qui a été intégré au sein l'architecture Tsunamy.

4.7 Conclusion

Au cours de ce chapitre, nous avons présenté l'ensemble de nos solutions concernant l'exécution sécurisée de plusieurs machines virtuelles sur une architecture manycore.

Nous avons d'abord présenté l'architecture Tsunamy qui a été développée au cours de nos travaux. Notre solution pour l'isolation des machines virtuelles consiste à l'ajout d'un troisième espace d'adressage, l'espace machine, et d'un composant matériel nommé *Hardware Address Translator* qui est en charge de la traduction des adresses machine en adresse physique. Ce composant est configuré au démarrage d'une machine virtuelle et ne peut être reconfiguré et désactivé qu'après l'arrêt de la machine virtuelle. Les HATs utilisent deux mécanismes de traduction suivant les adresses machines entrantes : si celles-ci ciblent un composant interne au cluster de la machine virtuelle, un mécanisme de traduction basé sur des opérations de translation est utilisé ; sinon, si les adresses machines ciblent un composant externe, typiquement un périphérique, un mécanisme semblable à la segmentation est mis en place. L'hyperviseur est lui aussi isolé des machines virtuelles par un HAT spécifique où

la configuration est fondue dans le matériel. Celle-ci contraint les accès de l'hyperviseur au cluster qu'il lui a été attribué, typiquement le cluster $(0, 0)$ de la plateforme. Notre solution concernant le partage des périphériques utilise la méthode d'assignation directe, consistant à offrir aux machines virtuelles un accès exclusif à des canaux de périphériques. Pour mettre en place cette solution nous avons dû développer deux périphériques multi-canaux, un contrôleur disque et un TTY. Nous avons aussi développé un HAT multi-canaux, situé derrière les périphériques externes, permettant de gérer plusieurs machines virtuelles, et donc d'effectuer différentes traductions en fonction de la machine virtuelle initiatrice de l'opération sur le périphérique.

Nous avons présenté notre procédure de démarrage d'une machine virtuelle qui consiste en l'exécution d'une procédure logicielle initiée par l'hyperviseur, nommée `startup_code`. L'exécution de cette procédure mène à la configuration des HATs par les cœurs de la machine virtuelle puis enfin à l'exécution du code `remote_hat_activation` permettant d'activer les HATs. La machine virtuelle s'isole donc de l'intérieur. Notre solution concernant l'arrêt des machines virtuelles met en jeu une procédure logicielle, nommée `shutdown_code`, ainsi que deux composants matériels, *Shutdown Virtual Machine Controller* et *Shutdown Virtual Machine Agent*. Ces composants effectuent une partie de la procédure d'arrêt et garantissent qu'à la fin de la procédure, les machines virtuelles sont bien arrêtées et que les ressources allouées à celle-ci sont bien libérées. Ces composants sont aussi en charge de désactiver les HATs de la machine virtuelle, permettant ainsi de pouvoir réutiliser les cœurs qui avaient été alloués.

Enfin, concernant la confidentialité des disques des machines virtuelles, nous avons mis en place une procédure de chiffrement et déchiffrement des images disques des machines virtuelles. Celles-ci contiennent le noyau du système d'exploitation et le système de fichiers. Les opérations de chiffrement et de déchiffrement de cette solution utilisent le crypto-processeur HCrypt intégré dans l'architecture.

Nous croyons que l'ensemble de ces solutions permet de répondre aux problématiques posées dans le chapitre 2.

4.8 Synthèse

La figure 4.31 illustre l'ensemble de nos solutions et la manière dont celles-ci interagissent. Y sont représentés les différents éléments de la TCB, aussi bien logiciels que matériels, ainsi que les éléments qui n'en font pas partie. Tous les éléments matériels sont dans la TCB et permettent la sécurisation de notre architecture. Certaines fonctions critiques de l'hyperviseur font aussi partie de la TCB, car elles permettent le bon fonctionnement du matériel, telle que la fonction de configuration des HATs et des SVMs. Les HATs sont des composants matériels qui permettent l'isolation des machines virtuelles et de l'hyperviseur et garantissent la confidentialité et l'intégrité de leurs données. Les SVMs sont les

composants en charge de l'arrêt des machines virtuelles et garantissent que les ressources allouées aux machines virtuelles sont libérées à la fin de la procédure d'arrêt. De plus, seuls ces derniers composants peuvent désactiver les HATs des machines virtuelles.

Nous pouvons voir sur la figure qu'il existe deux parties logicielles distinctes : une première qui s'exécute sur le cœur hyperviseur et une autre sur les cœurs de la machine virtuelle. La première est constituée de l'ensemble des fonctions de l'hyperviseur permettant le démarrage ou l'arrêt d'une machine virtuelle. Celles-ci sont séparées en deux parties : la première est le monde utilisateur qui peut recevoir les requêtes des utilisateurs, et l'autre est le noyau, qui permet de lancer les commandes de démarrage ou d'arrêt. Pour le logiciel s'exécutant sur les cœurs de la machine virtuelle, nous pouvons remarquer qu'il existe aussi deux parties distinctes : une première regroupant l'ensemble des fonctions hyperviseur réalisant la procédure logicielle de démarrage ou d'arrêt des machines virtuelles ; et une seconde partie concernant le code de la machine virtuelle elle-même, typiquement le système d'exploitation et les applications. La partie matérielle présentée sur la figure représente l'ensemble de la plateforme. Celle-ci contient principalement le cluster hyperviseur (qui comprend le cœur de l'hyperviseur et son HAT), et le cluster de la machine virtuelle (qui comprend les cœurs de la machine virtuelle, les HATs et le crypto-processeur HCrypt). Nous faisons aussi apparaitre les disques durs appartenant aux machines virtuelles et les objets logiciels qu'ils contiennent, à savoir le code noyau du système d'exploitation, le `bootloader` et le système de fichiers de la machine virtuelle.

Lorsqu'un utilisateur désire démarrer une machine virtuelle, celui-ci fait une demande sur le `Shell` de l'hyperviseur. Cette demande mène à l'exécution d'un `Syscall` permettant de basculer dans le mode noyau de l'hyperviseur. L'hyperviseur exécute alors des fonctions permettant l'allocation de ressources pour la nouvelle machine virtuelle et réveille le cœur *bootstrap*, ou cœur de démarrage, de la machine virtuelle à l'aide d'une WTI. La WTI fait sauter le cœur de démarrage dans le code spécifique du démarrage d'une machine virtuelle. Celui-ci exécute alors des fonctions spécifiques à la procédure de démarrage comme la configuration des HATs. Une fois la procédure logicielle de démarrage terminée les cœurs de la machine virtuelle exécutent le code de la machine virtuelle et plus particulièrement le `bootloader` du système d'exploitation. Le `bootloader` a la fonction de charger depuis le disque le noyau du système d'exploitation. Dans cette solution, celui-ci est chiffré et il est donc nécessaire de le déchiffrer avant de pouvoir l'exécuter. Pour cela, le `bootloader` utilise le crypto-processeur HCrypt présent dans l'architecture et copie dans sa mémoire le noyau déchiffré. Une fois le déchiffrement terminé, le `bootloader` peut exécuter le noyau du système d'exploitation de la machine virtuelle.

Lorsqu'un utilisateur veut arrêter une machine virtuelle, celui-ci le demande à l'hyperviseur via le `Shell`. Cette demande se traduit par une requête vers le SVM Controller situé dans le cluster hyperviseur. Le SVM Controller va alors transmettre l'ordre d'arrêt aux SVM Agents de la machine virtuelle qui vont par la suite activer le signal *soft reset* connecté aux différents cœurs du cluster. L'activation de ce signal permet de faire sauter les cœurs de la machine virtuelle au code d'arrêt des machines virtuelles. Ce code, représentant la procédure d'arrêt logicielle des machines virtuelles,

permet principalement d'assurer l'effacement des bancs mémoire alloués à la machine virtuelle. Une fois la mémoire effacée, les cœurs de la machine virtuelle s'endorment et le SVM Agent du cluster désactive alors les HATs de tous les clusters alloués à la machine virtuelle. Une fois tous les HATs désactivés, les SVMs Agent signalent au SVM Controller que la procédure d'arrêt est terminée. Les ressources alors anciennement allouées à la machine virtuelle sont libérées et peuvent être utilisées pour le déploiement d'une nouvelle machine virtuelle.

Nous pensons que l'ensemble des solutions présentées et que l'infrastructure décrite dans la figure 4.31 permet d'assurer l'exécution sécurisée de machines virtuelles sur une architecture many-core.

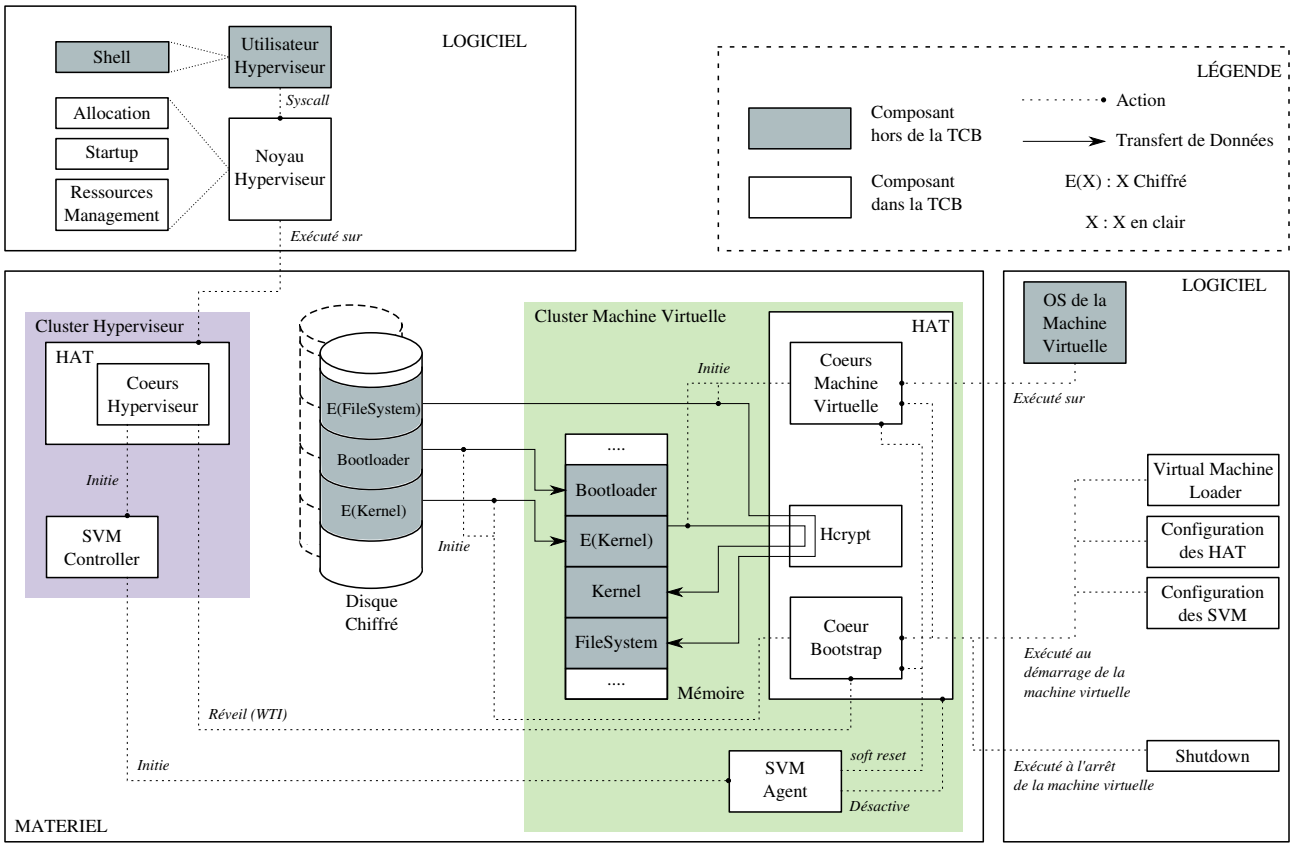

FIGURE 4.31 – Infrastructure Tsunamy

Chapitre

5 *Évaluations et résultats*

Contents

Ce chapitre vise à évaluer notre solution d'architecture manycore sécurisée pour l'exécution de plusieurs machines virtuelles. Cette évaluation est séparée en trois parties distinctes. Nous évaluerons tout d'abord dans la section 5.4 le surcoût matériel de notre solution, puis dans la section 5.5 le surcoût induit en performance. Pour cette évaluation, nous estimerons tout d'abord le surcoût induit par les HATs, le surcoût introduit par le système de fichiers chiffré et celui dû à l'exécution de plusieurs machines virtuelles concurrentes. Nous évaluerons également dans cette partie le temps pris par l'exécution de la procédure de démarrage et d'arrêt des machines virtuelles. Enfin, dans la section 5.6, nous ferons une analyse de la sécurité offerte par notre solution.

5.1 Plateforme expérimentale

Toutes nos évaluations ont été effectuées sur le modèle SystemC, précis au cycle, de l'architecture Tsunamy détaillée dans la section 4.1. Le prototype virtuel de cette architecture est générique et permet de générer des plateformes avec un nombre variable de clusters. Pour nos évaluations, nous avons fait varier le nombre de clusters entre 2 et 16. Nous avons donc aussi fixé le nombre maximal de machines virtuelles pouvant s'exécuter en parallèle à 16. Les paramètres des caches de premier et de second niveau sont donnés dans la table 5.1.

Paramètres	**Taille**
L1 Cache Sets (I & D)	64
L1 Cache Ways (I & D)	4
L1 Cache Words (I & D)	16
L2 Cache Sets (I & D)	256
L2 Cache Ways (I & D)	16
L2 Cache Words (I & D)	16
TLB Sets (I & D)	8
TLB Ways (I & D)	8

TABLE 5.1 – Paramètres des caches de l'architecture

5.2 Systèmes d'exploitation NetBSD et Almos

Toutes les évaluations ont été faites en utilisant le système d'exploitation ALMOS [16], qui est développé dans l'équipe. Celui-ci est un système d'exploitation expérimental de type UNIX dédié aux architectures manycore. Bien que le système d'exploitation NetBSD ait été porté sur notre solution, nous avons préféré utiliser ALMOS dans nos évaluations pour des raisons de temps de simulations. En effet, le temps requis pour le boot du système d'exploitation NetBSD est bien plus important que

celui nécessaire pour le système d'exploitation ALMOS.

Par construction, nos solutions ne sont pas dépendantes de la pile logicielle qui s'exécute au sein de la machine virtuelle. C'est pourquoi nous pensons que des résultats obtenus avec le système d'exploitation NetBSD auraient été les mêmes, en ordre de grandeur, que nos évaluations réalisées avec ALMOS.

Bien que nos évaluations ont été faites à l'aide du système d'exploitation ALMOS nous avons aussi exécuté le système d'exploitation NetBSD. Celui-ci nous a permis de valider les deux modes de traduction offerts par les HATs, à savoir les traductions pour des systèmes d'exploitation 40 et 32 bits. Par ailleurs, son portage a été très minimaliste, et a permis de montrer que la solution présentée n'est pas dépendante d'un système d'exploitation en particulier. Le portage a consisté à ajouter, au sein des drivers du contrôleur de disque de NetBSD, les instructions nécessaires pour la gestion des invalidations du cache L2. Pour cela avant chaque accès au disque nous invalidons, dans le cache L2, les adresses mémoire du buffer destination (ou source) utilisé par le driver. En effet, la version de NetBSD portée pour l'architecture TSAR ne prenait pas en compte le cluster I/O, et plus particulièrement la manière dont les périphériques, tel que le contrôleur de disque, communiquent avec la mémoire. Dans cette version les périphériques n'avaient pas d'accès direct aux bancs de mémoire, il n'était alors pas nécessaire de gérer la cohérence entre les caches L2 et la mémoire de façon logicielle.

Au cours de nos expérimentations nous avons également lancé simultanément des machines virtuelles exécutant ALMOS et NetBSD. Ceci nous a permis de montrer que notre hyperviseur et que les solutions matérielles sont vraiment indépendants des systèmes qui sont lancés et que ces deux systèmes peuvent cohabiter sur la même plateforme.

5.3 Choix des benchmarks

Les applications utilisées pour l'évaluation de nos solutions sont des applications parallèles *multi-threads*. Pour toutes les évaluations utilisant ces applications, nous plaçons un *thread* par cœur. Les applications utilisées peuvent se classifier en trois types suivant le modèle d'utilisation des données partagées :

- Les données ne sont pas partagées entre les *threads*.
- Les données sont partagées mais ne subissent pas de modifications par les *threads*.
- Les données sont partagées et subissent des modifications par les *threads*.

Pour nos évaluations nous avons utilisé quatre applications :

- `FFT` (*Fast Fourier Transform*) de la suite Splash-2 [106]. Cette application réalise la transformée de Fourier rapide. Les données sont partagées en lecture mais les modifications sont faites localement.

– `Histogram` de la suite de benchmarks Phoenix-2 [107]. Cette application produit un histogramme des pixels dans les canaux rouge, vert et bleu d'une image fournie en entrée. Cette image est découpée et distribuée entre chaque *thread* qui en traite localement une partie. Cette application ne possède pas de données partagées.
– `Kmeans` de la suite de benchmarks Phoenix-2. Cette application fait un partitionnement en k-moyennes qui est une méthode de partitionnement de données. Pour un nombre de points donnés et un entier k, on cherche à diviser les points en k sous-ensembles. Les données de cette application sont partagées en lecture mais les modifications sont faites localement.
– `Convol` est une application qui réalise un filtrage d'images médicales à l'aide d'un noyau de convolution. Le filtre est séparable et peut donc être réalisé en deux parties : un filtrage horizontal sur les lignes puis un filtrage vertical sur les colonnes. Le découpage en *threads* est fait de façon à ce que les lectures de données soient locales et les modifications soient distantes. De ce fait cette application appartient au troisième type présenté précédemment.

La table 5.2 présente la configuration de chaque application utilisée.

Application	**Données en entrée**
Histogram	image de 25 Mo (3 408 $\times$ 2 556)
Convol	image de 1 024 $\times$ 1 024 pixels
FFT	2^{18} points complexes
Kmeans	10 000 points

TABLE 5.2 – Paramètres des applications

5.4 Évaluation du surcoût matériel

Dans cette section nous cherchons à évaluer le surcoût matériel induit par l'ajout des composants HATs et des composants SVMs (Controller et Agent). Pour cela, nous comparons le nombre de bits mémorisant dans chacun des composants avec le nombre de bits mémorisant contenus dans un cache de premier niveau.

Comme précisé dans la section 4.2 les HATs possèdent 182 octets de mémoire. En comparaison, une ligne du cache de premier niveau contient 64 octets de mémoire. Sachant qu'il y a 256 lignes de caches dans un cache de premier niveau, et que le cache de premier niveau possède deux caches distincts (données et instructions), le coût total en bits mémorisant, sans compter le répertoire, est de 32Ko par cache de premier niveau. Notre plateforme comporte 4 cœurs par clusters, et donc 4 caches de premier niveau, soit un total de 128Ko. Notre solution nécessite 6 HATs par clusters : 4 au niveau des cœurs, 1 au niveau du DMA et 1 pour le HCrypt. Au total l'ajout des HATs équivaut donc à 1 092 octets de mémoire par clusters.

Le composant SVM Controller contient 24 octets de mémoire pour son fonctionnement interne et 5 octets de mémoire par canal, servant à stocker les informations de déploiement des machines virtuelles. Dans nos évaluations nous avons fixé le nombre de canaux à 16, puisque notre plus grande plateforme comportait 16 clusters. Dans ce cas d'étude, le composant SVM Controller possède donc 104 octets de mémoire.

Un composant SVM Agent contient 21 octets de mémoire pour son fonctionnement interne et 21 octets de mémoire pour les registres de configuration relatifs aux machines virtuelles. Au total un SVM Agent coûte donc 42 octets de mémoire. Sachant qu'il y a un SVM Agent par cluster, sauf dans le cluster hyperviseur qui contient le SVM Controller, et que nous nous plaçons dans un cas d'étude à 16 clusters, nous avons donc 15 SVM Agents soit un total de 630 octets de mémoire.

Nous pouvons en conclure que l'ajout des composants HATs, SVM Controller et SVM Agent a peu d'impact en terme de surcoût matériel. Par exemple, en considérant une plateforme à 16 clusters avec 4 cœurs par clusters, l'ensemble des caches de premier niveau représente 2Mo de mémoire alors que le cumul de nos composants sur l'ensemble de la plateforme représente 18Ko de mémoire soit moins de 1% de surcoût.

5.5 Évaluation du surcoût en performance

Dans cette section, nous visons à évaluer le surcoût en performance des solutions présentées dans le chapitre 4. Nous présenterons tout d'abord une évaluation du surcoût en performance en terme de temps d'exécution induit par l'introduction des composants HATs, puis du surcoût induit par le système de fichiers chiffré. Nous évaluerons aussi le temps nécessaire pour l'exécution des procédures de démarrage et d'arrêt d'une machine virtuelle. Enfin, nous évaluerons l'impact de l'exécution de plusieurs machines virtuelles de manière concurrentes sur une même plateforme.

5.5.1 Surcoût temporel induit par les HATs

La figure 5.1 montre le temps d'exécution sur la plateforme Tsunamy des 4 applications choisies. Ces temps sont normalisés par nombre de cœurs et par application par rapport aux temps d'exécutions de ces mêmes applications sur l'architecture TSAR.

Cette expérimentation vise à mesurer le surcoût en temps induit par l'ajout des HATs dans l'architecture, en particulier par la traduction supplémentaire effectuée pour toute requête sortant du L1. Chaque application a été exécutée sur des configurations à 1, 4, 8, 16 et 32 threads, où chaque thread a été déployé sur un cœur dédié. Pour toutes ces configurations, une seule machine virtuelle

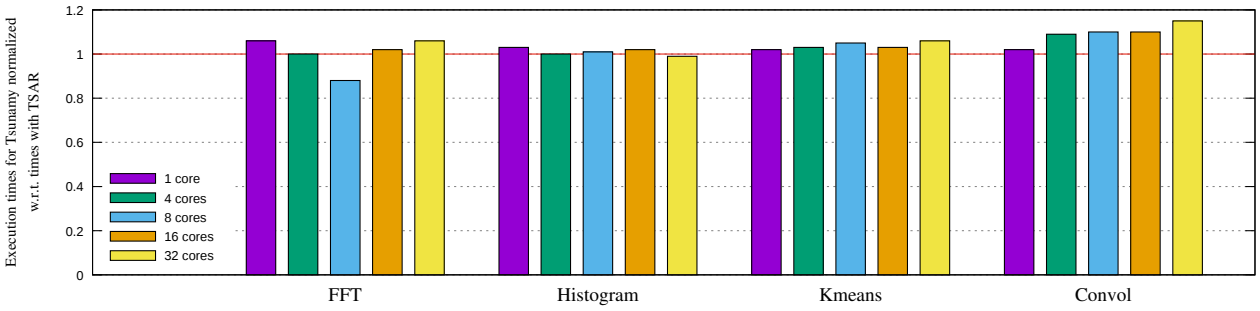

FIGURE 5.1 – Temps d'exécution de la phase parallèle des applications sur la plateforme Tsunamy normalisés par rapport aux temps sur la plateforme TSAR

est déployée sur la plateforme.

Pour cette évaluation, les temps mesurés correspondent à la phase parallèle des applications, car la façon dont les applications accèdent aux périphériques externes diffère entre la plateforme Tsunamy et la plateforme TSAR. De plus, mesurer les temps correspondant au chargement des fichiers d'entrées n'aurait pas de sens car les deux plateformes diffèrent sur la manière d'accéder aux périphériques externes. La plateforme Tsunamy possède un cluster I/O qui accède directement à la mémoire et nécessite une synchronisation des caches L2, alors que la plateforme TSAR utilisée ici n'en possède pas, et ses périphériques se trouvent directement dans le cluster 0. La raison pour laquelle la plateforme TSAR n'utilise pas de cluster I/O est que sa présence nécessite pour le système d'exploitation de gérer des adresses 40 bits. Or, le système d'exploitation ALMOS est un système 32 bits ne gérant pas les adresses physiques 40 bits. Cependant, la présence des HATs permet à des systèmes d'exploitation 32 bits de s'exécuter sur la plateforme Tsunamy, bien que celle-ci possède des adresses 40 bits.

De toutes les applications, le surcoût maximum est de 15% sur `Convol` sur 32 cœurs et le gain maximal se présente sur `FFT` sur 8 cœurs. Nous savons que l'ajout des HATs modifie les entrelacements des transactions et nous nous attendions que cela produise des petites variations, cependant nous ne savons pas comment un tel gain est possible.

Nous pouvons conclure que notre solution induit, en moyenne, une pénalité en performance de l'ordre de 3% pour toutes les différentes configurations et applications, ce que nous considérons comme acceptable compte tenu des garanties de sécurité apportées.

5.5.2 Surcoût temporel induit par le système de fichiers chiffré

Pour évaluer le surcoût induit par le système de fichier chiffré, nous avons réalisé un micro-noyau accédant à 1 000 pages consécutivement depuis le disque, et nous avons comparé le temps d'exécu-

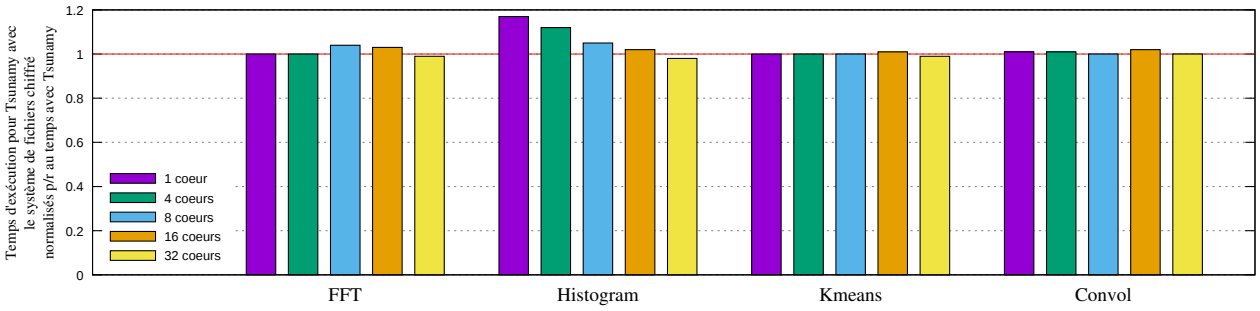

FIGURE 5.2 – Temps d'exécution pour le système de fichier chiffré normalisé par rapport aux temps avec un système de fichier classique

tion avec et sans le système de fichier chiffré. La résolution des défauts de page avec le système de fichier chiffré nécessite un déchiffrement de la page en utilisant le crypto-processeur HCrypt, qui peut traiter 128 bits de données en 11 cycles. Les résultats obtenus par ce micro-noyau montrent que la version avec le système de fichier chiffré est 1,3 fois plus lente. Cependant, nous pouvons noter que ce pire scénario est amplifié par le fait que le modèle de disque dur que nous utilisons ne possède pas de latence d'accès, ce qui rend la phase de déchiffrement bien plus importante, en proportion, dans la résolution d'un défaut de page, que ce qu'elle serait si le disque modélisé possédait une latence réaliste.

La figure 5.2 montre ce surcoût, mesuré en cycles processeur, pour les 4 applications `FFT`, `Histogram`, `Kmeans` et `Convol`. À cette fin, nous comparons pour une application et un nombre de cœurs donné, le temps total d'exécution avec et sans système de fichier chiffré. Chaque application est exécutée avec 1, 4, 8, 16, 32 threads et chaque thread est déployé sur un cœur dédié. Une fois encore, une seule machine virtuelle s'exécute sur la plateforme.

Pour les applications `FFT` et `Kmeans`, l'introduction du système de fichier chiffré n'impacte pas les performances. En effet, ces benchmarks ne font pas d'accès au disque dur mis à part pour les défauts de page liés aux instructions.

Pour `Convol`, le surcoût en performance induit par le système de fichier chiffré est d'en moyenne 1,5%. Même si cette application accède au disque dur pour charger l'image à traiter, dont la taille est de 2 Mo, le temps de chargement de l'image représente un faible pourcentage du temps de calcul de l'application, ce qui induit une faible dégradation des performances.

Pour `Histogram`, le surcoût induit par le système de fichier chiffré est plus important que pour les autres applications, car le chargement de l'image représente la majeure partie de cette application. En effet, le traitement effectué sur l'image est très limité. Nous pouvons noter que la dégradation en performance diminue avec le nombre de cœurs : cela est dû au fait que dans ALMOS, les accès au disque sont séquentialisés. Le surcoût induit par le déchiffrement du disque, d'environ 3 000 cycles par bloc de 4 096 octets, est très faible comparé au surcoût lié à la séquentialisation des accès disques.

Nous pouvons conclure que notre mécanisme de chiffrement du système de fichier n'impacte pas les applications qui ne font que peu d'accès au disque. Pour les applications utilisant le disque de manière plus intense, la dégradation en performance reste néanmoins acceptable.

5.5.3 Coût temporel de la procédure de démarrage d'une machine virtuelle

La figure 5.3 présente le temps d'exécution de la procédure de démarrage d'une machine virtuelle sur un nombre variable de clusters. Ce temps est mesuré depuis le début de la séquence, autrement dit lorsque la commande `run` est entrée dans le `shell` de l'hyperviseur, et jusqu'au réveil de tous les cœurs de la machine virtuelle.

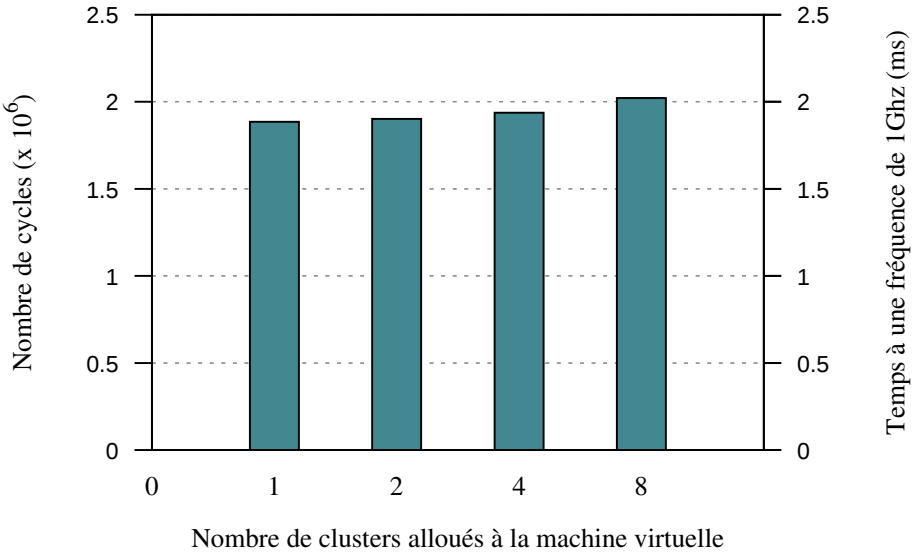

FIGURE 5.3 – Temps d'exécution du démarrage d'une machine virtuelle en fonction du nombre de clusters alloués

Le temps moyen nécessaire au démarrage d'une machine virtuelle est de 1,9 millisecondes [1] pour un processeur possédant une fréquence de fonctionnement de 1 GHz. Nous pouvons noter que le temps pris par la séquence de démarrage augmente légèrement avec le nombre de clusters alloués à la machine virtuelle. Ceci peut être expliqué par le fait que plus le nombre de clusters alloués à une machine virtuelle est important, plus la procédure de démarrage doit configurer de composants matériels.

Sur la figure 5.4, nous pouvons noter que 95% du temps de la séquence de démarrage est consacrée à la phase exécutant du code de la machine virtuelle, qui comprend l'exécution du code du `preloader` et du `bootloader`. Le `preloader` a pour principal but de charger depuis le disque dur, le code du `bootloader` du système d'exploitation de la machine virtuelle. Le `bootloader` réalise principalement le chargement depuis le disque du noyau du système d'exploitation, son déchiffrement,

1. Ce chiffre n'est pas réaliste dans le sens où les accès au disque n'ont pas de latence

puis l'initialisation de structures de données. Le temps pris par ces étapes est indépendant du nombre de clusters alloués à une machine virtuelle, sauf pour la phase d'initialisation des structures de données, dont le temps augmente lorsque le nombre de clusters augmente.

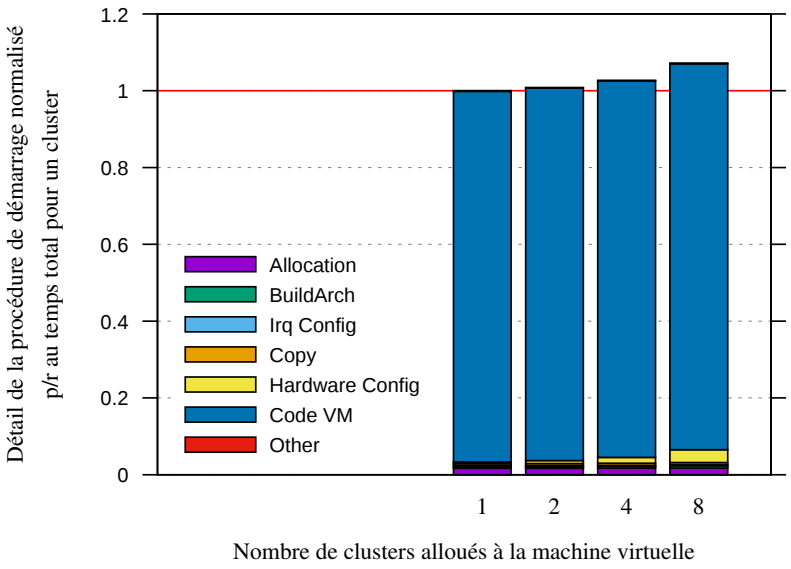

FIGURE 5.4 – Détail du temps passé dans les phases du démarrage d'une machine virtuelle par rapport au nombre de clusters alloués

Dans la figure 5.5, nous avons supprimé les étapes liées à l'exécution du code de la machine virtuelle, et nous pouvons remarquer que la phase de configuration du matériel (qui représente un pourcentage très faible du temps total sur la figure 5.4) est dépendante du nombre de clusters alloués.

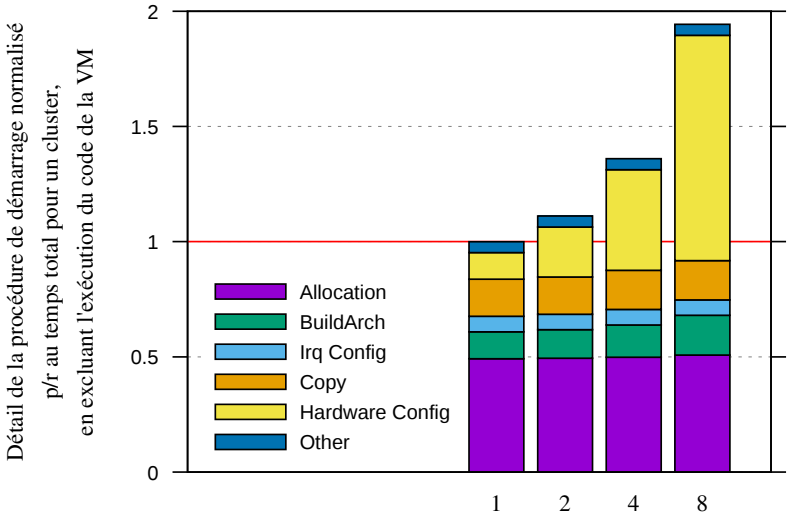

FIGURE 5.5 – Détail du temps passé dans les phases du démarrage d'une machine virtuelle, en excluant les phases d'exécution du code de la machine virtuelle, par rapport au nombre de clusters alloués

Nous pouvons donc conclure que le temps pris par le démarrage des machines virtuelles est indépendant de la taille de la machine virtuelle, mis à part pour la phase de configuration du matériel, mais qui représente une très faible part de la procédure de démarrage. La majorité du temps nécessaire pour démarrer une machine virtuelle est prise par la phase d'exécution du code de la machine virtuelle, comprenant l'exécution du `preloader` et du `bootloader` du système d'exploitation de la machine virtuelle.

5.5.4 Coût temporel de la procédure d'arrêt des machines virtuelles

La figure 5.6 montre le temps d'exécution de la procédure d'arrêt d'une machine virtuelle déployée sur un nombre variable de clusters. Ce temps est mesuré depuis le début de la séquence d'arrêt, autrement dit lorsque la commande `stop` est entrée sur le `shell` de l'hyperviseur, jusqu'à la réception par l'hyperviseur de l'interruption signalant l'arrêt de la machine virtuelle. Pour ces expérimentations, nous avons utilisé des RAMs de 64 Mo pour chaque cluster.

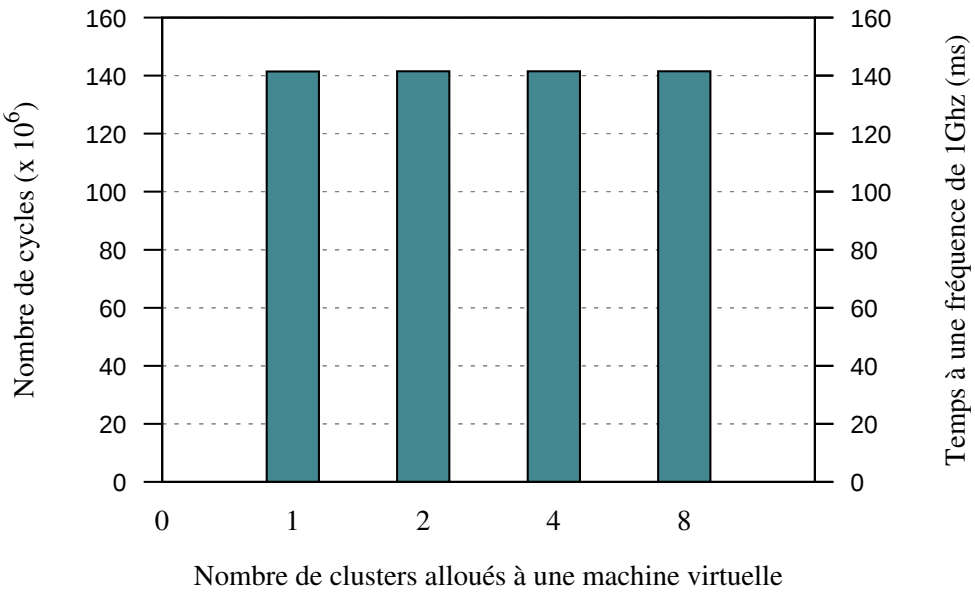

FIGURE 5.6 – Temps nécessaire pour l'exécution de la procédure d'arrêt d'une machine virtuelle en fonction du nombre de clusters alloués

Le temps moyen pour arrêter une machine virtuelle est de 140 millisecondes. Nous pouvons remarquer que le mécanisme d'arrêt des machines virtuelles est indépendant de la taille de celle-ci, autrement dit que le nombre de clusters alloués à une machine virtuelle n'influe pas sur le temps de la séquence d'arrêt. Ceci s'explique par le fait que nous avons conçu une procédure d'arrêt distribuée, dans laquelle tous les clusters travaillent en parallèle. En revanche, le temps du mécanisme d'arrêt est très dépendant de la taille des RAMs de chaque cluster, car la remise à zéro des bancs de mémoire utilisés par la machine virtuelle représente 99,9% du temps de la procédure d'arrêt. Nous pouvons

donc dire que le temps pris par le mécanisme d'arrêt est proportionnel à la capacité des RAMs. La taille de RAM choisie pour ces expérimentations est clairement en dessous de la taille d'une vraie mémoire : 1 Go de mémoire par cluster serait plus réaliste. Néanmoins, comme cette phase est déjà parallèle et utilise les DMAs, la seule façon d'accélérer cette étape serait d'utiliser des mémoires possédant une extension matérielle de remise à zéro. Le détail du temps passé dans les différentes étapes de la procédure d'arrêt n'est pas donné dans la figure 5.6 car seule l'étape d'effacement des RAMs serait visible.

Nous pouvons conclure que le mécanisme d'arrêt des machines virtuelles est indépendant de la taille de la machine virtuelle mais dépendant de la quantité de mémoire présente dans les clusters de la plateforme. La majorité du temps nécessaire pour arrêter une machine virtuelle est prise par la phase d'effacement des bancs de mémoire.

5.5.5 Évaluation de l'impact de la cohabitation de machines virtuelles

Ces expérimentations visent à déterminer si l'exécution concurrente de plusieurs machines virtuelles peut mener à une dégradation en performance des machines virtuelles, à cause du partage du réseau global. Avec les contraintes de topologie appliquées au déploiement des machines virtuelles, nous nous attendons à ne pas avoir de dégradation si les machines virtuelles ne font pas d'accès aux périphériques externes. Pour vérifier cette hypothèse, nous avons exécuté 15 machines virtuelles en parallèle, une par cluster, exécutant toutes la même application sur 4 cœurs. Nous avons ensuite comparé les temps d'exécution avec celui correspondant à la même application, exécutée également sur une machine virtuelle déployée sur un cluster, mais s'exécutant seule sur la plateforme. La figure 5.7 montre ces temps d'exécution pour toute l'application alors que la figure 5.8 montre seulement les temps d'exécution de la phase parallèle de l'application.

Ces résultats mettent en avant le fait qu'en l'absence d'accès I/O intensifs, les temps d'exécution restent exactement les mêmes. La phase d'initialisation de l'application `Histogram`, qui consiste au chargement d'une image, crée beaucoup de contention sur le contrôleur de disque, qui ne possède qu'un seul port initiateur et cible. En effet, la copie en mémoire de tous les blocs des 15 images est faite de manière séquentielle. De plus, la phase de chargement représente une portion non négligeable de l'application, même en mono-thread. Ces résultats étaient attendus, et pour pallier ce point de contention, il est nécessaire d'avoir soit plusieurs contrôleurs de disque, soit un contrôleur de disque multi-ports.

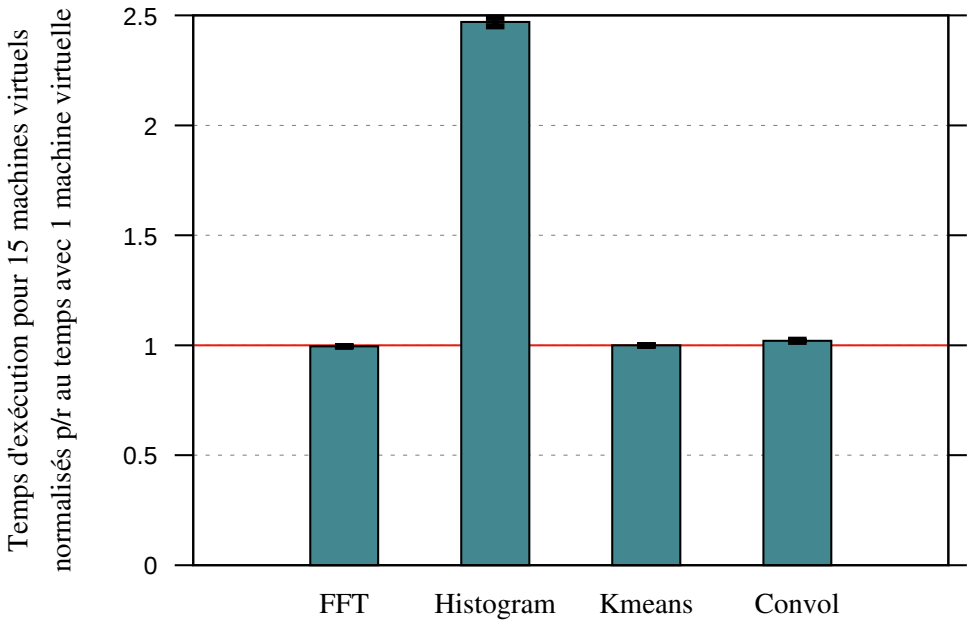

FIGURE 5.7 – Temps moyen de l'ensemble de l'exécution de 15 applications s'exécutant en parallèle (1 par machine virtuelle), comparé à une application s'exécutant seule sur la plateforme. Les barres d'erreur représentent l'écart type pour les 15 machines virtuelles.

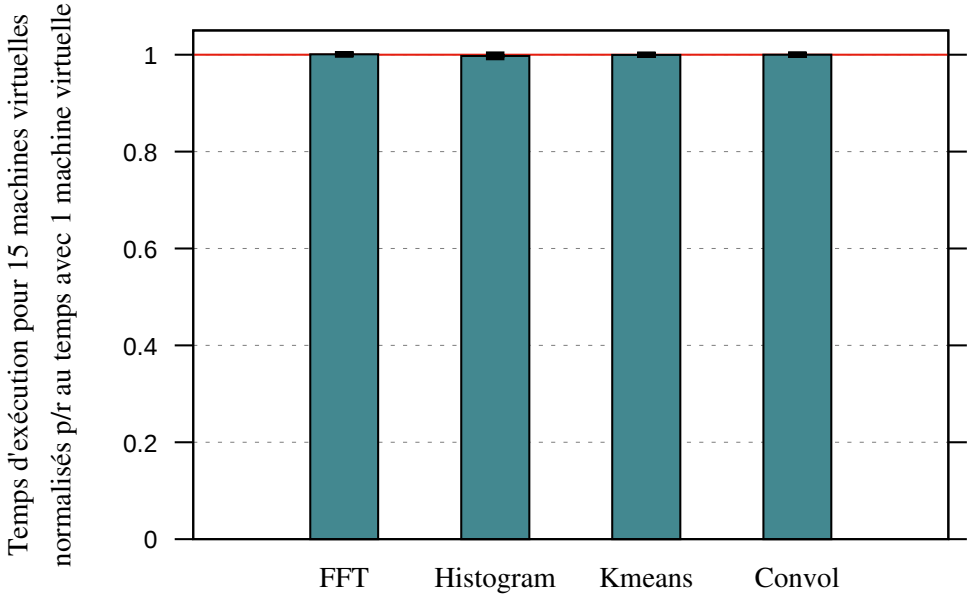

FIGURE 5.8 – Temps moyen d'exécution de la phase parallèle de 15 applications s'exécutant en parallèle (1 par machine virtuelle), comparé à une application s'exécutant seule sur la plateforme. Les barres d'erreur représentent l'écart type pour les 15 machines virtuelles.

5.6 Évaluation de la sécurité

Cette section vise à analyser la sécurité apportée par l'architecture sécurisée Tsunamy. Dans un premier temps nous présenterons le modèle de menace considéré ainsi que les différents acteurs mis en jeu. Nous présenterons ensuite notre analyse des vulnérabilités et des différentes attaques possibles.

5.6.1 Modèle de menace

Tout d'abord, il convient de différencier trois propriétés de sécurité :
- La confidentialité et l'intégrité.
- La pérennité des données.
- La garantie de service.

Le but de notre travail est de protéger l'intégrité et la confidentialité des machines virtuelles s'exécutant sur la plateforme. Les propriétés de pérennité des données et de garantie de service ne sont pas des priorités pour nous et celles-ci ne sont pas nécessairement garanties. Pour chacune des attaques nous énoncerons quelles propriétés sont préservées.

Dans la solution présentée, nous traitons uniquement les attaques logicielles, autrement dit des attaques provenant d'un code malicieux (dues par exemple à l'exploitation d'une faille de sécurité) pouvant s'exécuter sur une machine virtuelle ou sur l'hyperviseur, et visant soit à déranger le bon fonctionnement de machines virtuelles s'exécutant sur la plateforme, soit à en récupérer des informations. Notre architecture ne cherche pas non plus à se prémunir de vulnérabilités internes aux machines virtuelles, autrement dit qu'une faille de sécurité dans une machine virtuelle puisse corrompre cette même machine virtuelle.

Les composants matériels présents dans l'architecture sont considérés comme sûrs et nous faisons l'hypothèse qu'ils ont le comportement attendu, autrement dit que ceux-ci ne possèdent pas de failles de sécurité pouvant mener à l'accomplissement d'une attaque sur les machines virtuelles. Cependant, une fuite des données par une lecture des registres est possible et le logiciel est en charge d'assurer une remise à zéro des registres des composants lorsque ceux-ci ne sont plus alloués à une machine virtuelle.

La figure 5.9 présente les différents acteurs dans notre analyse de sécurité et les interactions qu'ils possèdent.

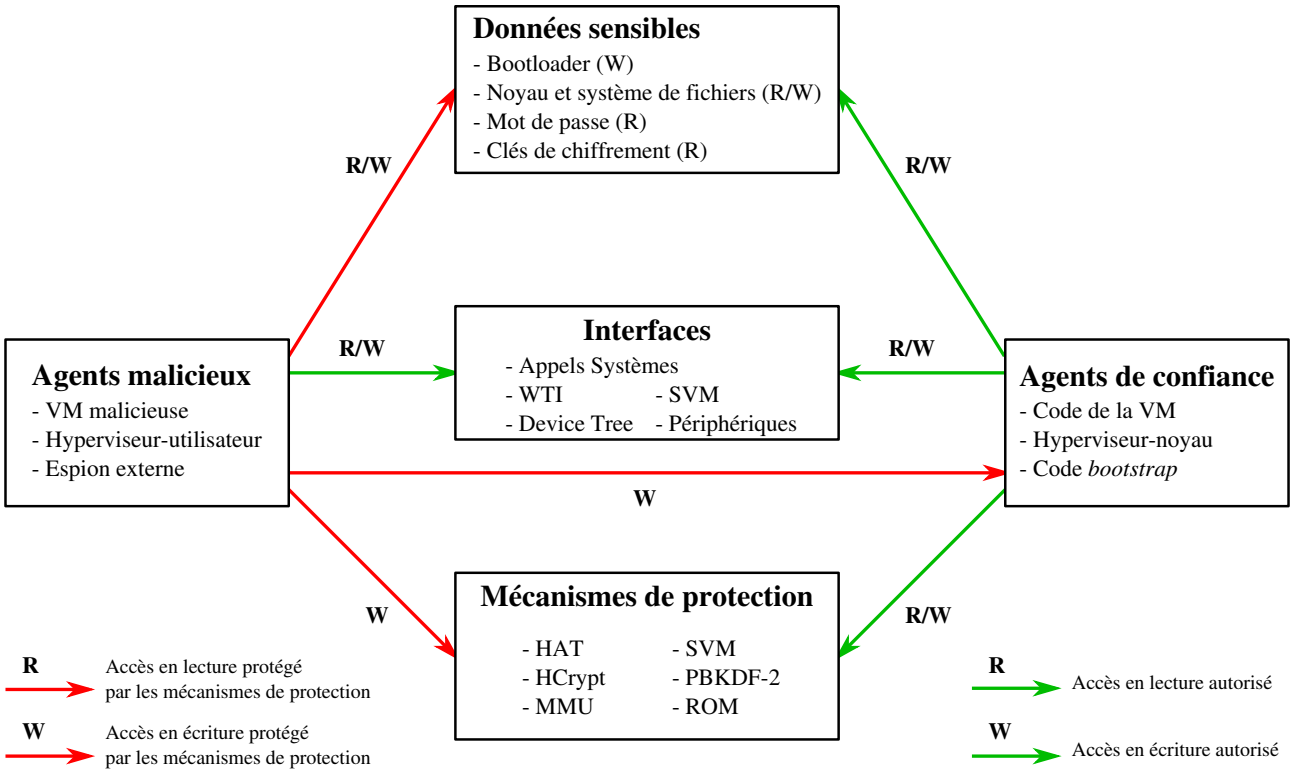

FIGURE 5.9 – Analyse de sécurité : Acteurs et interactions

Agents malicieux

Les agents malicieux sont les acteurs qui essayent de mettre en péril l'intégrité ou la confidentialité des agents de confiance. Nous considérons trois types d'agents malicieux : une machine virtuelle malicieuse, le code utilisateur de l'hyperviseur et un espion externe.

Une machine virtuelle malicieuse est une machine virtuelle qui exécute du code malicieux ayant pour but d'attaquer une autre machine virtuelle. Nous pouvons considérer trois cas de machine virtuelle malicieuse. Le premier cas est une machine virtuelle s'exécutant en concurrence avec sa machine virtuelle cible et qui essaye d'accéder aux informations de sa cible ; ce cas est appelé *VM MALICIEUSE CONCURRENTE*. Le deuxième cas est une machine virtuelle qui a été exécutée juste avant et sur les mêmes clusters que la machine virtuelle cible. Cette machine virtuelle malicieuse peut alors avoir placée dans la mémoire de la machine virtuelle cible du code malicieux mettant en péril la sécurité de la machine virtuelle cible ; ce cas est appelé *VM MALICIEUSE PRÉDÉCESSEURE*. Enfin le troisième cas est une machine virtuelle s'exécutant après la machine virtuelle cible et sur les mêmes clusters. Cette machine virtuelle peut alors essayer de récupérer en mémoire ou dans des périphériques des informations restantes ; ce cas est appelé *VM MALICIEUSE SUCCESSEURE*.

le code utilisateur de l'hyperviseur, nommé `hyperviseur-utilisateur`, correspond au code s'exécutant dans la partie utilisateur de l'hyperviseur. Ce code peut présenter des vulnérabilités per-

mettant de corrompre certaines fonctionnalités de l'hyperviseur et mener à des attaques sur l'hyperviseur ou sur les machines virtuelles s'exécutant sur la plateforme.

Un espion externe correspond à un attaquant qui peut accéder à des données d'une machine virtuelle à travers les périphériques ou à des données stockées en dehors de la puce. Typiquement cela peut correspondre à un attaquant pouvant modifier le contenu des disques durs d'une machine virtuelle.

Agents de confiance

Les agents de confiance sont les entités qui sont autorisées à accéder aux données sensibles ou qui sont responsables de la configuration des mécanismes de protection. Nous faisons l'hypothèse que les différentes entités en charge de cette configuration ne présentent pas de failles de sécurité et fonctionnent correctement. Les différents agents de confiance sont listés ci-dessous :

- Le code de la machine virtuelle, à savoir le `bootloader`, le noyau et les applications, qui sont de confiance pour les données de la même machine virtuelle.
- Le code vérifié de l'hyperviseur, nommé code `bootstrap` dans la suite du document.
- Le code de la partie noyau de l'hyperviseur, nommé code `hyperviseur-noyau` dans la suite du document.

Nous considérons le code de la machine virtuelle comme étant un agent de confiance pour elle-même, c'est à dire qu'il possède les autorisations nécessaires pour accéder aux données sensibles de la machine virtuelle. Néanmoins le code de la machine virtuelle ne possède pas les accès nécessaires à la configuration des mécanismes de protection.

Le code `bootstrap` représente les fonctions critiques permettant de mettre en place les mécanismes de protection. Ces fonctions sont listées ci-dessous :

- Le code de `reset` de l'hyperviseur permettant de mettre en place la séparation noyau/utilisateur, et donc la configuration de la MMU de l'hyperviseur.
- L'ensemble de la procédure de démarrage des machines virtuelles et plus particulièrement les fonctions en charge de la configuration des HATs et des SVMs.
- L'ensemble de la procédure logicielle d'arrêt des machines virtuelles et plus particulièrement la fonction permettant de remettre à zéro la mémoire utilisée par la machine virtuelle.

Le code `hyperviseur-noyau` correspond au code s'exécutant dans la partie noyau de l'hyperviseur. Même si ce code est de confiance, s'il possèdait une ou plusieurs failles, leur exploitation resterait limitée. En effet, la propriété de garantie de service ne serait plus assurée, mais les propriétés d'intégrité, de confidentialité et de pérennité des données seraient préservées. En ce sens, une faille dans ce code n'est pas nécessairement critique d'un point de vue sécurité.

Données sensibles

Les données sensibles représentent les données de la machine virtuelle qui doivent être protégées. Ci-dessous, nous listons les différents éléments que nous considérons comme des données sensibles, ainsi que leurs garanties de sécurité nécessaires :

- Le code et les données du système d'exploitation de la machine virtuelle. Ces éléments doivent être protégés en confidentialité et en intégrité.
- Le système de fichiers du système d'exploitation de la machine virtuelle comprenant les données utilisateur et les applications. Cet élément nécessite une protection en confidentialité et en intégrité.
- Le code et les données du `bootloader` du système d'exploitation. Ces éléments doivent être protégés en intégrité.
- Le mot de passe utilisateur, utilisé pour déchiffrer l'image disque. Il nécessite une protection en confidentialité.
- Les clés de chiffrement du disque, à savoir K_1 et la clé K_2 utilisée pour la chiffrer, qui doivent être protégées en confidentialité.

Ces données sensibles sont généralement stockées dans la mémoire dédiée à la machine virtuelle, mais peuvent aussi être présentes dans la mémoire externe, comme par exemple dans le disque dur. Les clés de chiffrement sont stockées dans la mémoire du HCrypt. Comme décrit dans la section 4.6 les clés de chiffrement sont chiffrées par le HCrypt à l'aide d'une clé maître. Ainsi, en dehors du HCrypt, ces clés circulent sous forme chiffrée. Tous les HCrypt de la plateforme utilisent la même clé maître, il est donc nécessaire de pouvoir assurer la sécurité du chiffré des clés K_1 et K_2 car un agent malicieux pourrait alors charger ces clés dans un HCrypt qui lui a été alloué et ainsi déchiffrer le disque dur d'une autre machine virtuelle.

L'intégrité du mot de passe et des clés de chiffrement ne présente pas un danger, en terme de confidentialité et d'intégrité, pour la machine virtuelle. En effet une modification de ces données mènerait uniquement à un déni de service car le `bootloader` serait alors incapable de s'authentifier et donc de déchiffrer son image disque.

Mécanismes de protection

Les mécanismes de protection sont les contre-mesures mises en place permettant d'assurer la confidentialité et l'intégrité des données sensibles. Nous considérons ici six mécanismes de protection :

- Les HATs qui permettent l'isolation et le partitionnement des ressources des machines virtuelles et de l'hyperviseur, en ajoutant un espace d'adressage.
- Les composants SVMs utilisés dans la procédure d'arrêt des machines virtuelles qui sont en

charge de remettre à zéro la mémoire utilisée par les machines virtuelles après leur arrêt.
- Les mécanismes de cryptographie qui permettent de chiffrer le contenu du disque des machines virtuelles, et plus particulièrement le cryptoprocesseur HCrypt qui réalise les opérations de chiffrement, déchiffrement ainsi que la génération et la gestion des clés.
- La fonction cryptographique PBKDF-2 qui est utilisée pour protéger le mot de passe utilisateur et pour produire la clé (K_2) utilisée pour chiffrer la clé du disque (K_1).
- Les différents modes de privilèges du processeur (Utilisateur/Noyau) ainsi que la MMU, qui permettent la séparation de l'hyperviseur en deux parties utilisateur et noyau.
- La ROM qui contient le code de l'hyperviseur, rendant ainsi impossible toute modification de ce code.

Nous faisons l'hypothèse que tous ces mécanismes répondent à leur spécification et ne présentent aucune vulnérabilité.

Interface de communication entre les agents de confiance et malicieux

Cette interface représente les mécanismes que peuvent utiliser les agents malicieux pour communiquer avec les agents de confiance. Nous pouvons lister quatre interfaces : {hyperviseur noyau, hyperviseur utilisateur}, {hyperviseur noyau, bootstrap}, {bootstrap, VM} et {espion externe, VM}.

Pour l'interface {hyperviseur noyau, hyperviseur utilisateur}, la partie utilisateur de l'hyperviseur communique avec la partie noyau via des appels systèmes. Ces deux parties peuvent échanger des données à l'aide de registres ou de mémoire partagée.

L'interface {hyperviseur noyau, bootstrap} possède les mécanismes suivants :
- Les WTIs envoyées par l'hyperviseur noyau lors de la procédure de démarrage, qui permettent le réveil des cœurs d'une machine virtuelle.
- Une zone de mémoire partagée utilisée par les cœurs de démarrage pour récupérer des informations nécessaires au démarrage de la machine virtuelle, typiquement les informations d'allocation de clusters.
- Une requête au SVM Controller lors de la procédure d'arrêt d'une machine virtuelle, qui va alors déclencher l'exécution du code `bootstrap` sur les cœurs de la machine virtuelle.

L'interface {`bootstrap`, VM} est limitée aux informations que le code `bootstrap` fournit aux machines virtuelles – typiquement le *device-tree*.

L'interface {espion externe, VM} entre un espion externe et les machines virtuelles est possible car un espion externe peut interagir avec les machines virtuelles par le biais des périphériques externes. En particulier, un espion externe pourrait venir lire ou écrire des données sur le disque d'une machine

virtuelle.

Nous pouvons aussi noter qu'il n'existe pas de moyen de communication entre les machines virtuelles, mis à part en utilisant des périphériques comme un contrôleur Ethernet. De plus l'hyperviseur étant aveugle et désengagé, celui-ci ne possède aucune interface avec le code des machines virtuelles.

5.6.2 Analyse des vulnérabilités

Dans cette sous-section nous allons vérifier que les différents acteurs malicieux ne peuvent pas modifier ou accéder aux mécanismes de protection. Ensuite, nous vérifierons que chaque acteur malicieux ne peut pas accéder aux données sensibles des agents de confiance.

Protections des mécanismes de protection

L'hyperviseur, ainsi que le code des machines virtuelles, ne peuvent pas accéder aux HATs à cause de la configuration des HATs des cœurs exécutant ces objets logiciels. En effet, les registres adressables des HATs ne sont pas *mappés* dans l'espace accessible par l'hyperviseur et les machines virtuelles. De plus une fois que les HATs sont configurés par le code `bootstrap`, ceux-ci ne peuvent plus être reconfigurés. Seuls les SVM Agents, lors de la procédure d'arrêt, peuvent débloquer les HATs. Ainsi, même un bug dans l'allocation des clusters qui menerait au lancement d'une machine virtuelle (2) sur des clusters déjà utilisés par une machine virtuelle (1), ne pourrait mener à la corruption des données de la machine virtuelle (1). En effet, le code `bootstrap` se verrait retourner une erreur matérielle par les HATs lors de la phase de configuration des HATs. Un tel bug mènerait alors à une défaillance de l'hyperviseur (arrêt de son exécution causé par une erreur interne), mais pas à une perte de confidentialité et d'intégrité des données de la machine virtuelle (1).

L'hyperviseur et les machines virtuelles peuvent accéder uniquement aux crypto-processeurs HCrypt présents dans les clusters qui leur sont alloués. En effet, la configuration des HATs permet d'interdire l'accès aux registres adressables des HCrypt présents dans les autres clusters.

La partie utilisateur de l'hyperviseur ne peut pas modifier la protection mise en place par la MMU. En effet, le matériel assure que seul un logiciel s'exécutant dans le mode le plus privilégié peut modifier les registres de la MMU. De plus, la table des pages, située dans la mémoire de l'hyperviseur, est à la fois protégée du monde utilisateur via la MMU, mais aussi des accès en provenance des machines virtuelles par le biais de la configuration des HATs. Les cœurs des machines virtuelles ne peuvent bien évidemment pas modifier les registres de configuration de la MMU de l'hyperviseur, vu qu'aucun cœur ne peut être partagé entre l'hyperviseur et le code d'une machine virtuelle.

Les composants SVM Controller et SVM Agent ne peuvent voir leurs configurations modifiées au cours de la vie d'une machine virtuelle, ainsi la procédure d'arrêt assure bien que l'ensemble de la machine virtuelle est arrêtée et une faille dans l'hyperviseur ne peut pas mener à l'exécution d'une procédure d'arrêt partielle, par exemple dans laquelle seule la moitié des clusters d'une machine virtuelle serait libérée. Cette protection est assurée par la configuration des HATs des machines virtuelles et de l'hyperviseur.

Le composant ROM n'étant pas configurable par nature, nous sommes assurés qu'aucune modification par des attaques logicielles de son contenu n'est possible.

Les algorithmes de cryptographie utilisés par le crypto-processeur HCrypt et la fonction PBKDF-2 sont supposés robustes.

Attaques par une machine virtuelle malicieuse

La figure 5.10 montre les actions que pourrait entreprendre une machine virtuelle malicieuse envers les différentes données sensibles et les agents de confiance.

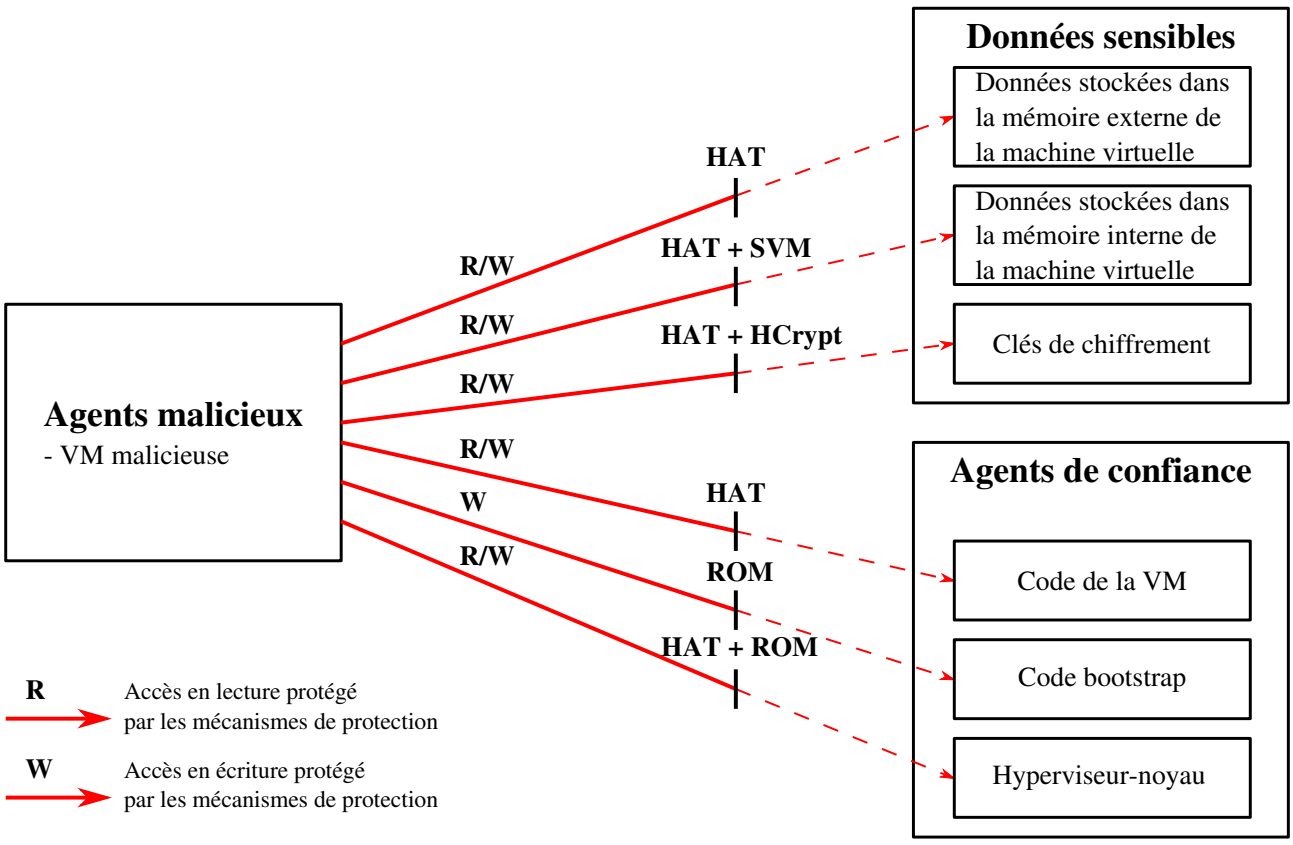

FIGURE 5.10 – Protections contre les attaques d'une machine virtuelle malicieuse

Une machine virtuelle concurrente pourrait vouloir accéder à des données contenues dans la mémoire allouée à une machine virtuelle cible ou dans la partie noyau de l'hyperviseur, ou alors modifier le code de la machine virtuelle cible. Or, la protection offerte par les HATs permet de restreindre l'accès d'une machine virtuelle malicieuse aux ressources qui lui ont été allouées.

Une machine virtuelle successeure ne peut pas accéder à des données sensibles d'une ancienne machine virtuelle car la procédure d'arrêt remet à zéro la mémoire allouée à l'ancienne machine virtuelle. Par le même mécanisme, une machine virtuelle prédecesseure ne peut pas laisser du code malicieux en mémoire dans le but d'altérer le comportement d'une future machine virtuelle.

Une machine virtuelle ne peut pas accéder à des données d'une machine virtuelle cible stockées dans des mémoires externes, par exemple à des données stockées dans le TTY ou le disque dur, car les HATs permettent d'assurer un accès exclusif à des canaux de périphériques.

Une machine virtuelle malicieuse ne peut pas modifier le code de l'hyperviseur ou le code `bootstrap`, car ceux-ci se trouvent en ROM.

Attaques ciblant la partie utilisateur de l'hyperviseur

La figure 5.11 montre les différentes actions que pourrait entreprendre un attaquant dans le cas d'une corruption de la partie utilisateur de l'hyperviseur.

La partie utilisateur de l'hyperviseur ne peut pas accéder aux ressources allouées aux machines virtuelles, aussi bien la mémoire dédiée que les périphériques externes. Cette interdiction est assurée par la présence, et la configuration, des HATs du cluster hyperviseur.

Si elle était corrompue, la partie utilisateur de l'hyperviseur pourrait accéder à certaines données sensibles stockées dans la mémoire de l'hyperviseur noyau, comme par exemple les tables d'allocation de clusters. Dans cette hypothèse, cela lui permettrait de corrompre l'allocation des ressources pour une nouvelle machine virtuelle. Or, la protection offerte par la MMU permet d'assurer qu'un logiciel s'exécutant dans le mode utilisateur de l'hyperviseur ne puisse accéder à des données du mode noyau.

Enfin, la partie utilisateur de l'hyperviseur ne peut pas modifier le code `bootstrap`, ni le code noyau de l'hyperviseur, car ceux-ci se situent en ROM.

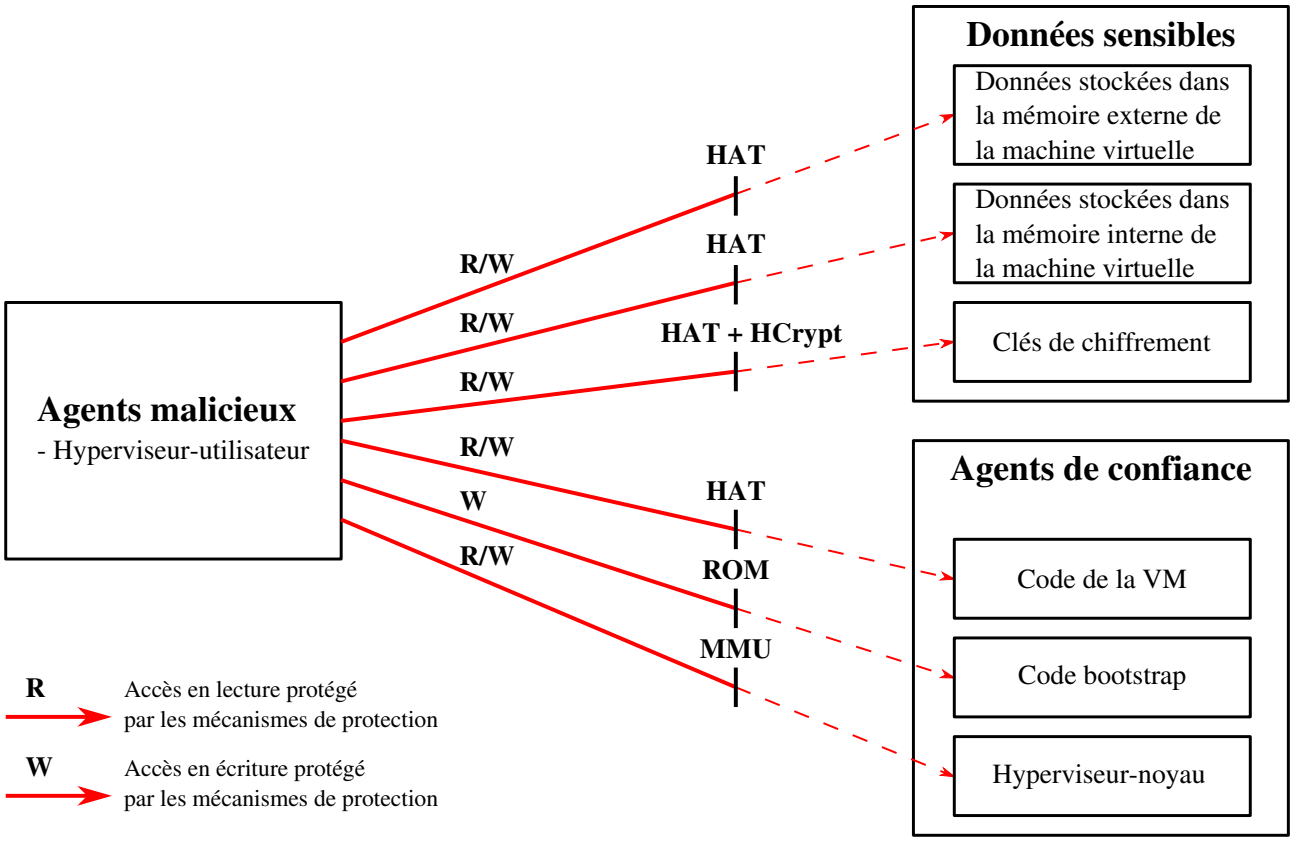

FIGURE 5.11 – Protections contre les attaques de la partie utilisateur de l'hyperviseur

Attaques sur les périphériques externes

Dans le cas d'une attaque sur les périphériques externes, nous considérons plus particulièrement un accès en lecture ou en écriture du disque dur d'une machine virtuelle. Ces accès ne peuvent pas être faits de façon logicielle, mais uniquement par un accès physique aux disques durs. Les systèmes de fichiers et noyaux présents sur les disques durs étant chiffrés, un accès en lecture ne compromet pas la propriété de confidentialité, et un accès en écriture ne compromet pas la propriété d'intégrité. Une modification du disque peut néanmoins mener à une perte de données des machines virtuelles cibles, et donc compromettre la propriété de pérennité des données.

En revanche, le code du `bootloader` n'étant pas chiffré, celui-ci est complètement sensible à une lecture ou à une écriture. Une lecture, dans notre cas, ne présente pas de danger puisque le code du `bootloader` ne possède pas de données sensibles ; mais une modification du `bootloader`, en particulier un remplacement du `bootloader`, pourrait mener à une attaque de type *phishing* permettant de faire fuiter de manière non visible le mot de passe utilisateur entré. L'attaquant pourrait ainsi déchiffrer le disque dur de la machine virtuelle cible.

Notre implémentation actuelle ne présente pas de contre-mesure contre cette vulnérabilité,

mais une solution à base d'authentification du code du `bootloader` pourrait permettre de vérifier que le `bootloader` du système d'exploitation de la machine virtuelle n'a pas été modifié par un attaquant externe.

Surface d'attaque et complexité

La table 5.3 présente la taille, en terme de lignes de code, des différentes parties de l'hyperviseur. Il contient un total de 5 000 lignes de code, en excluant les commentaires, ce qui respecte la propriété de petite empreinte. Notre hyperviseur est séparé en trois parties majeures : la partie noyau, la partie utilisateur, et le code `bootstrap`. Dans notre travail, le code `bootstrap` est la seule partie qui nécessite impérativement d'être vérifiée, car ce code contient les procédures logicielles de démarrage et d'arrêt des machines virtuelles, et plus particulièrement les fonctions faisant la configuration du matériel (HATs et SVMs). Ces procédures logicielles possèdent en moyenne 350 lignes de code chacune. Comme nous l'avons vu, bien que nous considérons la partie noyau comme étant de confiance, la présence d'un bug dans cette partie ne compromettrait pas la confidentialité, l'intégrité et la pérennité des données, mais pourrait mener à une défaillance de l'hyperviseur. Concernant la librairie `libfdt` utilisée pour la génération du *device-tree*, celle-ci a été portée telle quelle pour utilisation par l'hyperviseur. Nous pensons néanmoins que sa taille pourrait être réduite.

Fonction	**LoC**
Noyau	1936
Procédure de démarrage	305
Procédure d'arrêt	400
Libfdt	1337
Shell	965
Total	4943

TABLE 5.3 – Complexité de l'hyperviseur en lignes de code (LoC)

Déni de service

L'architecture manycore sécurisée Tsunamy ne cherche pas à répondre en priorité aux problématiques de disponibilité des machines virtuelles.

Pour assurer une garantie de service, il serait nécessaire de faire confiance à l'intégralité du noyau de l'hyperviseur, en plus du code `bootstrap`. Cela est envisageable dans le sens où la TCB serait toujours relativement petite. De plus, de part l'allocation de ressources dédiées aux machines virtuelles, aussi bien des cœurs que de la mémoire ou des périphériques, nous limitons par construction les

possibilités de déni de service sur ces ressources. Le réseau sur puce, qui est partagé par toutes les machines virtuelles, a malgré tout un partage limité pour les accès internes des machines virtuelles, du fait de la politique d'allocation convexe des clusters. Seuls les accès aux périphériques peuvent créer un ralentissement des machines virtuelles, car bien que les périphériques soient multi-canaux, les ports de réception et d'émission de requêtes de ces composants sont souvent uniques. De plus, l'accès au cluster I/O étant unique, une certaine contention peut alors avoir lieu.

Néanmoins il est aussi important de noter que par construction le réseau sur puce utilisé dans notre architecture implémente une politique d'arbitrage *round-robin* permettant d'assurer que les requêtes provenant de différentes entités seront toujours servies et qu'il ne peut pas y avoir de famine.

Enfin, grâce aux très faibles interférences entre les différentes machines virtuelles, nous avons réduit les possibilités d'attaques par canaux cachés. Par exemple, des attaques via l'analyse de l'usage des caches est impossible puisque les machines virtuelles ne partagent aucun cache. Néanmoins, il est possible d'imaginer un scénario dans lequel une machine virtuelle malicieuse ou l'hyperviseur essaierait d'espionner les accès au réseau sur puce des machines virtuelles, en mesurant les temps d'accès aux périphériques. Cependant, cela nous semble très compliqué à mettre en place et peu fiable, pour un résultat n'apportant que peu d'informations.

En ce qui concerne la pérennité des données, il faut ajouter un mécanisme de sauvegarde régulier des disques afin de limiter les risques liés à la corruption des disques des machines virtuelles.

5.6.3 Conclusion

Dans cette section, nous avons analysé la sécurité de notre architecture. Celle-ci permet d'exécuter plusieurs machines virtuelles en offrant des garanties de confidentialité et d'intégrité pour toutes les entités s'exécutant sur la plateforme. Nous avons aussi montré que la *Trusting Computing Base*, ainsi que la surface d'attaque, restent faibles. Nous avons identifié des vulnérabilités en ce qui concerne la disponibilité des machines virtuelles et plus particulièrement concernant une attaque où l'attaquant aurait un accès physique au disque dur.

5.7 Conclusion

Nous avons évalué dans ce chapitre les solutions proposées dans cette thèse. Nous avons tout d'abord estimé le coût matériel de notre solution, puis la dégradation en performance induite par les modifications apportées à l'architecture TSAR. Pour finir, nous avons analysé la sécurité apportée par notre solution.

Notre estimation du coût matériel de notre solution montre que celui-ci est négligeable : en effet, le cumul des composants rajoutés, en termes de bits mémorisant, sur une plateforme à 16 clusters représente à peine 1% de la mémoire requise pour l'ensemble des caches de premier niveau de la plateforme.

En termes de dégradations des performances, la présence des HATs induit en moyenne une perte de 3% sur le temps d'exécution des applications, ce qui est acceptable au vu des garanties de sécurité apportées. Nous avons aussi montré que l'introduction d'un système de fichiers chiffré présente une dégradation acceptable, grâce au HCrypt. Nos expérimentations à l'aide d'un micro-noyau réalisant 1 000 accès au disque présente une dégradation de 30% par rapport à la version sans système de fichiers chiffré. Il est cependant important de noter que notre modèle actuel de contrôleur de disque accède au disque sans latence, contrairement à la réalité, ce qui amplifie la dégradation apportée par le déchiffrement du disque.

Nous avons aussi évalué l'interférence des machines virtuelles quand celles-ci s'exécutent de façon concurrente. Nos expérimentations montrent que dans le cas d'accès internes, typiquement des accès à la mémoire dédiée aux machines virtuelles, aucune dégradation n'est constatée. Néanmoins, dans le cas d'accès aux périphériques, nous avons remarqué une dégradation en performance, celle-ci étant principalement due à une contention sur le périphérique. Nous avons aussi évalué le temps requis et la scalabilité des procédures de démarrage et d'arrêt des machines virtuelles. Nous avons remarqué que les deux procédures possèdent un temps d'exécution quasi constant, autrement dit que celui-ci ne change pas en fonction de la taille de la machine virtuelle.

Notre analyse de la sécurité de la solution apportée montre que celle-ci apporte un haut niveau de sécurité concernant la confidentialité et l'intégrité des machines virtuelles et de l'hyperviseur. Nous avons montré qu'un agent malicieux ne peut pas accéder aux données sensibles de l'hyperviseur ou des autres machines virtuelles et qu'il ne peut pas venir modifier les configurations des composants en charge de la protection de ces données. Notre analyse a mis en évidence une vulnérabilité concernant le disque dur, dans le cas où l'attaquant possède un accès physique permettant de modifier son contenu. Plus particulièrement, il peut récupérer un mot de passe lié à un système d'exploitation cible s'il peut remplacer le `bootloader` correspondant. Notre solution n'adresse pas spécifiquement la propriété de disponibilité des machines virtuelles, mais l'architecture offre par construction certaines garanties à ce niveau. Enfin, la *Trusting Computing Base*, ainsi que la surface d'attaque de notre système est relativement petite, car la partie critique de l'hyperviseur, c'est-à-dire le code nécessitant d'être vérifié, possède environ 700 lignes.

Chapitre

6 Conclusions et perspectives

Dans cette thèse nous avons essayé de répondre à trois questions portant sur la sécurisation de l'exécution de machines virtuelles sur des architectures manycore à espace d'adressage partagé de type CC-NUMA. Pour cela nous avons développé une architecture nommée Tsunamy basée sur l'architecture de référence TSAR. Nos contributions, à la fois matérielles et logicielles, permettent de garantir des propriétés de confidentialité et d'intégrité pour les machines virtuelles et l'hyperviseur s'exécutant sur la plateforme.

6.1 Conclusion

Cohabitation de machines virtuelles

La première question portait sur la cohabitation des machines virtuelles et sur l'exécution de systèmes d'exploitation invités non modifiés. Notre solution consiste à ajouter des composants au niveau des modules initiateurs sur le réseau sur puce et à ajouter un niveau d'adressage. Ces composants, nommés *Hardware Address Translator* (HAT), sont en charge d'effectuer la traduction de ce troisième niveau d'adressage, et donc de pouvoir utiliser la virtualisation complète, et de garantir une isolation stricte entre les différentes machines virtuelles et l'hyperviseur. Le mécanisme de traduction de ces composants utilise des informations de topologie pour les accès aux périphériques répliqués, typiquement la mémoire ou des DMAs locaux, et une table de segment pour les accès aux périphériques externes, comme le contrôleur de disque.

Les évaluations ont montré que le surcoût matériel, inférieur à 1%, est négligeable. De plus, nous avons aussi montré que l'impact sur les performances de l'introduction des HATs dans l'architecture est faible. Pour cela, nous avons mesuré une dégradation sur le temps d'exécution de quatre

applications de l'ordre de 3% en moyenne. Concernant les interférences entre machines virtuelles, nous avons montré qu'en dehors d'une forte utilisation des périphériques, les machines virtuelles ne subissent aucune dégradation en s'exécutant en parallèle.

Limitation des droits de l'hyperviseur

La deuxième question portait sur la restriction des droits d'accès de l'hyperviseur sur les ressources allouées à une machine virtuelle. Pour cela, nous avons implémenté des *Hardware Address Translators* particuliers dans le cluster hyperviseur, qui ne possèdent pas d'interface de configuration. La configuration concernant l'isolation de l'hyperviseur est alors fondue dans les composants et il est impossible de venir adresser ces composants. La configuration des *Hardware Address Translators* Hyperviseur cantonne l'hyperviseur dans le cluster $(0, 0)$ de la plateforme. L'hyperviseur se voit attribuer un accès à certains périphériques externes, tels que le TTY ou la BROM. Néanmoins, l'hyperviseur peut venir adresser n'importe quelle XICU de la plateforme, afin de pouvoir réveiller les processeurs des machines virtuelles lorsqu'il doit démarrer une nouvelle machine virtuelle.

Nous avons noté que la présence d'un *Hardware Address Translators* Hyperviseur, dont la configuration est fixe, présente une perte de généricité pour la plateforme. La plateforme n'est alors plus utilisable sans hyperviseur, ou alors dans un mode dégradé où le cluster $(0, 0)$ est inutilisable. Une solution logicielle pourrait pallier cette perte de généricité, en utilisant les mêmes HATs que dans les clusters des machines virtuelles et en mettant en place une procédure logicielle effectuant la configuration de ces composants pour le cluster hyperviseur au démarrage de la machine.

Nous avons aussi proposé une procédure de chiffrement des disques durs des machines virtuelles permettant ainsi de garantir la confidentialité des données persistantes des machines virtuelles. Pour cela, nous utilisons le crypto-processeur Hcrypt intégré dans l'architecture Tsunamy. Ce crypto-processeur est en charge de chiffrer et déchiffrer les blocs à chaque opération sur le contrôleur de disque. L'impact en performance induit par l'intégration d'un système de fichiers chiffré est acceptable. En effet, le chiffrement, comme le déchiffrement, réalisé par le crypto-processeur met en moyenne 3 000 cycles pour des blocs de 4 096 octets. En comparaison au temps d'accès des disques, nous estimons que ce surcoût temporel est acceptable au vue des garanties de sécurité offertes.

Déploiement et arrêt des machines virtuelles

La troisième et dernière question portait sur le déploiement et l'arrêt dynamique des machines virtuelles s'exécutant sur la plateforme. Pour cela, notre solution propose des mécanismes de démarrage et d'arrêt des machines virtuelles basés sur des procédures logicielles assistées par le matériel.

Notre mécanisme de démarrage des machines virtuelles est initié par l'hyperviseur mais les étapes concernant la sécurité des machines virtuelles, comme la configuration des HATs ainsi que leur activation, sont réalisées par les cœurs de la machine virtuelle elle-même. Ceux-ci exécutent un code de confiance, nommé `startup_code`, qui permet de configurer les composants liés à la sécurité, après quoi ils exécutent une fonction, nommée `RHA`, permettant l'activation de leur HAT. Les machines virtuelles sont alors isolées de l'intérieur et seul le `startup_code` nécessite d'être vérifié. Nous avons aussi évalué notre procédure de démarrage, en terme de temps d'exécution, et avons noté que celle-ci est indépendante du nombre de clusters alloués à la machine virtuelle. De plus, la majorité du temps prise par la procédure de démarrage est liée à l'exécution du `preloader` et du `bootloader` du système d'exploitation invité.

Le mécanisme de démarrage assure la bonne configuration du matériel d'isolation et s'effectue par un logiciel de confiance s'exécutant sur les cœurs de la machine virtuelle. Ainsi, l'hyperviseur n'est pas impliqué dans la configuration du matériel.

Le mécanisme d'arrêt d'une machine virtuelle vise à proposer une interface, accessible aussi bien par l'hyperviseur que par la machine virtuelle elle-même. Notre mécanisme s'appuie sur deux composants matériels et sur une procédure logicielle. Les deux composants sont en charge d'assurer le bon fonctionnement de la procédure d'arrêt et sont appelés : *Shutdown Virtual Machine Controller* (SVM Controller) et *Shutdown Virtual Machine Agents* (SVM Agent). Le composant SVM Controller est le composant maître vu par l'hyperviseur, et sert à initier la procédure d'arrêt. Il délègue à ses agents l'arrêt des machines virtuelles. Le composant SVM Controller se trouve dans le cluster hyperviseur alors que les SVM Agents sont répliqués dans tous les clusters réservés aux machines virtuelles. Ces composants enclenchent la procédure d'arrêt et assurent qu'à la fin de celle-ci, la machine virtuelle est bien arrêtée et que la mémoire allouée à la machine virtuelle est effacée. Ces composants sont aussi les seuls à pouvoir désactiver les HATs, étape nécessaire avant le re-déploiement d'une nouvelle machine virtuelle.

Les évaluations concernant la procédure d'arrêt montrent que celle-ci est complètement distribuée et donc indépendante du nombre de clusters alloués à une machine virtuelle. Néanmoins, la procédure d'arrêt est dépendante de la quantité de mémoire disponible dans chaque cluster et la quasi-totalité du temps nécessaire pour arrêter une machine virtuelle est prise par l'effacement de ces bancs mémoire.

6.2 Perspectives

Authentification du Boot

Comme nous l'avons montré dans l'analyse de la sécurité de notre solution, celle-ci possède une faille importante concernant le remplacement du `bootloader` si un attaquant possède un accès physique sur le disque. En effet, celui-ci étant en stocké en clair, il existe une possibilité pour un attaquant de le remplacer afin de se faire passer pour le `bootloader` et ainsi permettre la récupération du mot de passe de l'utilisateur. Notre solution actuelle ne tient pas compte de ce problème et nous avons supposé que le code du `bootloader` ne pouvait pas être remplacé. Cependant, il est bien sûr difficile de ne pas tenir compte de cette faille et nous pensons que le problème d'un démarrage sécurisé est important. Une méthode pourrait consister à chiffrer le `bootloader` mais il faudrait alors que l'hyperviseur le déchiffre, et donc cela voudrait dire que l'hyperviseur doit connaitre le mot de passe de l'utilisateur, ce qui est difficilement concevable dans un contexte d'hypervision en aveugle. Une autre solution, qui de notre point de vue parait plus abordable, serait d'authentifier le code du `bootloader` pour permettre à l'utilisateur d'être assuré de communiquer avec son `bootloader` lors de l'entrée du mot de passe. Une solution à base de *Trusted Platform Module* (TPM), qui est un standard permettant d'assurer entre autre l'authentification de la plateforme, pourrait être envisagée. Nous pourrions intégrer au sein des composants ajoutés les propriétés nécessaires au TPM, par exemple au sein du crypto-processeur HCrypt.

Au même titre que le `bootloader`, le noyau et le système de fichiers du système d'exploitation invité ne sont pas authentifiés, bien que chiffrés. Il serait alors aussi possible pour un attaquant de modifier leur contenu en modifiant des blocs du disque. Une telle modification aurait un comportement indéfini, puisque ces objets logiciels sont stockés sous forme chiffrée. Pour prévenir ce genre d'attaque, il faudrait ajouter au contrôleur de disque une extension matérielle permettant de stocker des meta-données par bloc. Celles-ci pourraient contenir un *hash* du bloc, qui serait re-calculé par le contrôleur de disque à chaque accès au bloc. Cette vulnérabilité ne nous parait que très peu pertinente car nous ne pensons pas qu'une attaque visant cette faille puisse aboutir à un résultat pertinent pour l'attaquant, par exemple à une récupération d'informations sensibles. Au mieux, nous pensons que ce type d'attaques peut aisément provoquer des exceptions dans le noyau du système d'exploitation invité et donc un déni de service sur la machine virtuelle.

Séparation de l'hyperviseur et mécanisme de récupération

Actuellement, l'hyperviseur n'est pas séparé selon les deux modes noyau et utilisateur. Nous pensons cependant que cette propriété est importante et nécessite d'être réalisée. En effet, la séparation noyau/utilisateur de l'hyperviseur permet de renforcer la robustesse de notre solution. Actuelle-

ment, rien n'empêche un *bug* dans les fonctions non critiques, telle que la console, de venir corrompre des données importantes. Par exemple, un *buffer overflow* dans la console pourrait modifier les tables d'allocation des machines virtuelles et mener à un déploiement d'une nouvelle machine virtuelle sur des clusters déjà alloués. Bien que nous avons montré qu'un tel scénario ne menaçait pas la confidentialité et l'intégrité des machines virtuelles, cela peut mener à un plantage de l'hyperviseur. La séparation du noyau et de l'utilisateur au sein de l'hyperviseur pourrait aussi permettre de mettre en place un système de récupération pour les services. Par exemple, si la console connait un dysfonctionnement, nous pourrions relancer le service plutôt que de devoir redémarrer l'hyperviseur (et donc toute la plateforme actuellement). Au même titre, il serait intéressant d'implémenter ce service dans les fonctions critiques de l'hyperviseur, comme la procédure de démarrage, car actuellement un *bug* dans le démarrage d'une machine virtuelle, par exemple une exception liée à une mauvaise configuration, mène au plantage de l'hyperviseur et nous oblige à redémarrer l'ensemble du système.

Suppression des MULTI_HAT

Actuellement, nous avons ajouté des MULTI_HAT devant chaque périphérique externe possédant une capacité DMA. Dans notre plateforme, cela se limite à un seul composant, le contrôleur de disque, ce qui n'est donc pas problématique. Dans le cas d'une plateforme plus réaliste, où le nombre de périphériques DMAs risque d'être plus important, cela peut devenir coûteux car il nous faudrait alors ajouter plus de MULTI_HAT. De plus, nous aimerions que notre solution se cantonne à la modification de ce qui se trouve à l'intérieur de la puce et non de ce qui se trouve à l'extérieur, comme le cluster I/O. Pour résoudre ce problème, nous pouvons retirer les MULTI_HAT et intégrer ses services dans le composant IOB. Comme toutes les requêtes émises par les périphériques passent par l'IOB, nous pouvons donner la capacité à ce composant de réaliser le service de traduction. Cela nous permettrait alors de ne pas répliquer les composants MULTI_HAT mais juste de complexifier un peu le composant IOB en lui ajoutant un ensemble de jeu de registres permettant de réaliser les traductions. Cette solution nécessite néanmoins de pouvoir étiqueter les requêtes émises par les périphériques, à savoir d'ajouter dans les requêtes l'identifiant de la machine virtuelle, pour permettre à l'IOB l'utilisation du bon jeu de registres. Avec cette solution, nous devons aussi interdire les communications directes entre deux périphériques, puisque le service de traduction est situé dans l'IOB et non plus devant les périphériques.

Intégration d'une pile IP et infrastructure *cloud computing*

Dans l'implémentation actuelle, les communications entre l'hyperviseur et l'utilisateur sont émulées par l'utilisation du composant MULTI_TTY_VT. Nous avons pleinement conscience que cette solution n'est pas réaliste et nécessite d'être améliorée. Pour cela, il serait intéressant d'intégrer dans notre plateforme un contrôleur Ethernet multi-canaux et de porter une pile réseau au sein de l'hy-

perviseur, ainsi que dans notre système d'exploitation ALMOS. Nous pourrions alors mettre en place une infrastructure de *Cloud Computing* où les utilisateurs communiquent avec le serveur à distance, et donc l'hyperviseur ainsi que les machines virtuelles, par le biais du réseau. Il serait alors aussi important de porter tous les services nécessaires au sein de l'hyperviseur et cela ferait nécessairement grandir la surface d'attaque de notre l'hyperviseur, c'est pourquoi nous pensons que cette étape doit être réalisée après avoir implémenté la séparation du noyau de l'hyperviseur. Ce type d'infrastructure pose nécessairement des problèmes de sécurité, car il faut s'assurer que les communications soient sécurisées, et que les données de l'utilisateur soient toujours protégées de l'hyperviseur. Il serait aussi nécessaire de modifier un peu la procédure de démarrage puisque l'image d'une machine virtuelle serait potentiellement transmise par le réseau. Il existe dans la littérature de nombreuses solutions pour ces problèmes et il serait intéressant de voir si elles sont compatibles avec le concept d'hypervision aveugle.

Dissémination des sources et publications

Le code source des composants matériels ainsi que celui de l'hyperviseur est disponible sur le dépôt SVN du projet ANR Tsunamy [108]. Ce code est placé sous licence *open-source* : *GNU Lesser General Public License*.

Ces travaux de thèse ont mené à deux publications. Une première publication [109] a eu lieu dans la conférence internationale *2015 Nordic Circuits and Systems Conference* (NORCAS'15). Cette publication décrit principalement le fonctionnement des *Hardware Address Translator* et une évaluation préliminaire du surcoût induit par l'ajout de ces composants. Un deuxième article [110] a été publié dans la revue *Microprocessors and Microsystems : Embedded Hardware Design* (MICPRO) comme une extension de la publication à la conférence NORCAS'15. Ce deuxième article contient la description de l'intégralité des contributions proposées par cette thèse.

Bibliographie

[1] Statista, "Size of the cloud computing and hosting market market worldwide from 2011 to 2019 (in billion U.S. dollars)," https://www.statista.com/statistics/500541/worldwide-hosting-and-cloud-computing-market/.

[2] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming : Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, 2012.

[3] J. F. Levine, J. B. Grizzard, and H. L. Owen, "Detecting and categorizing kernel-level rootkits to aid future detection," *IEEE Security Privacy*, 2006.

[4] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

[5] K. Kortchinsky, "Cloudburst : Hacking 3d (and breaking out of vm-ware). url : https ://www. blackhat. com/presentations/bh-usa-09/kortchinsky," *BHUSA09-Kortchinsky-Cloudburst-SLIDES. pdf (vid. pág. 13)*, 2009.

[6] P. Stewin and I. Bystrov, "Understanding dma malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2012.

[7] S. Borkar, "Thousand core chips : a technology perspective," in *Proceedings of the 44th annual Design Automation Conference*. ACM, 2007, pp. 746–749.

[8] D. Woo and H. Lee, "Extending amdahl's law for energy-efficient computing in the many-core era," vol. 41, no. 12. IEEE Computer, Dec. 2008.

[9] P. Guerrier and A. Greiner, "A generic architecture for on-chip packet-switched interconnections," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '00, 2000.

[10] L. Benini and G. De Micheli, "Networks on chips : a new soc paradigm," *Computer*, vol. 35, no. 1, 2002.

[11] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel, "A distributed run-time environment for the kalray mppa®-256 integrated manycore processor," *Procedia Computer Science*, vol. 18, 2013, 2013 International Conference on Computational Science.

[12] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal, "Atac : A 1000-core cache-coherent processor with on-chip optical network," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10, 2010.

[13] T. Corporation, http://www.tilera.com.

[14] I. Corporation, "SCC External Architecture Specification," https://communities.intel.com/docs/DOC-5044/version.

[15] LIP6 and BULL, "TSAR (Tera-Scale Architecture)," https://www-soc.lip6.fr/trac/tsar.

[16] LIP6, "Advanced Locality Management Operating System," http://www.almos.fr.

[17] R. J. Creasy, "The origin of the vm/370 time-sharing system," *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 483–490, 1981.

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

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

[20] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, "Intel virtualization technology for directed i/o." *Intel technology journal*, vol. 10, no. 3, 2006.

[21] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor." in *USENIX Annual Technical Conference, General Track*, 2001, pp. 1–14.

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

[23] A. Whitaker, M. Shaw, and S. D. Gribble, "Scale and performance in the denali isolation kernel," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 195–209, 2002.

[24] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz, "Unmodified device driver reuse and improved system dependability via virtual machines." in *OSDI*, vol. 4, no. 19, 2004, pp. 17–30.

[25] K. Mansley, G. Law, D. Riddoch, G. Barzini, N. Turton, and S. Pope, "Getting 10 gb/s from xen : safe and fast device access from unprivileged domains," in *European Conference on Parallel Processing*. Springer, 2007, pp. 224–233.

[26] J. R. Santos, Y. Turner, G. J. Janakiraman, and I. Pratt, "Bridging the gap between software and hardware techniques for i/o virtualization." in *USENIX Annual Technical Conference*, 2008, pp. 29–42.

[27] D. E. Bell and L. J. LaPadula, "Secure computer systems : Mathematical foundations," DTIC Document, Tech. Rep., 1973.

[28] C. E. Landwehr, "Formal models for computer security," *ACM Computing Surveys (CSUR)*, vol. 13, no. 3, pp. 247–278, 1981.

[29] K. J. Biba, "Integrity considerations for secure computer systems," DTIC Document, Tech. Rep., 1977.

[30] D. D. Clark and D. R. Wilson, "A comparison of commercial and military computer security policies," in *Security and Privacy, 1987 IEEE Symposium on*. IEEE, 1987, pp. 184–184.

[31] M. Bishop, "Introduction to computer security."

[32] C. Instruction, "4009,"national information assurance glossary," committee on national security systems, may 2003," *Formerly NSTISSI*, vol. 4009, 2003.

[33] F. PUB, "Standards for security categorization of federal information and information systems," 2004.

[34] D. C. Latham, "Department of defense trusted computer system evaluation criteria," *Department of Defense*, 1986.

[35] P. Dubrulle, R. Sirdey, P. Dore, M. Aichouch, and E. Ohayon, "Blind hypervision to protect virtual machine privacy against hypervisor escape vulnerabilities," in *Industrial Informatics (INDIN), 2015 IEEE 13th International Conference on*, 2015.

[36] C. Dall and J. Nieh, "Kvm/arm : The design and implementation of the linux arm hypervisor," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM, 2014, pp. 333–348.

[37] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm : the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, 2007.

[38] R. J. Masti, C. Marforio, K. Kostiainen, C. Soriente, and S. Capkun, "Logical partitions on many-core platforms," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC 2015, 2015.

[39] Y. Zhang, W. Pan, Q. Wang, K. Bai, and M. Yu, "HypeBIOS : Enforcing VM Isolation with Minimized and Decomposed Cloud TCB." http://www.people.vcu.edu/~myu/s-lab/my-publications.html.

[40] J. Szefer and R. B. Lee, "Architectural support for hypervisor-secure virtualization," in *SIGARCH Comput. Archit. News*, 2012.

[41] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick, "Xen 3.0 and the art of virtualization," in *Linux symposium*, vol. 2. Ottawa, Ontario, Canada, 2005, pp. 65–78.

[42] VMware, "VMware vSphere Hypervisor," https://www.vmware.com/fr/products/vsphere-hypervisor.html/.

[43] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel : A new os architecture for scalable multicore systems," in *In Symposium on Operating Systems Principles*, ser. SOSP'09, 2009.

[44] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, "Helios : Heterogeneous multiprocessing with satellite kernels," in *In Symposium on Operating Systems Principles*, ser. SOSP'09, 2009.

[45] M. Lamine Karaoui, "Système de fichiers scalable pour architectures many-core) faible empreinte énergétique." Ph.D. dissertation, Université Pierre et Marie Curie (UPMC), 2016.

[46] W. Shi, J. Lee, T. Suh, D. H. Woo, and X. Zhang, "Architectural support of multiple hypervisors over single platform for enhancing cloud computing security," in *In Computing Frontiers*, ser. CF'12, 2012.

[47] Y. Dai, Y. Qi, J. Ren, Y. Shi, X. Wang, and X. Yu, "A lightweight vmm on many core for high performance computing," ser. SIGPLAN Not, 2013.

[48] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," in *In Computer and Communications Security*, ser. CCS'11, 2011.

[49] R. Wojtczuk, J. Rutkowska, and A. Tereshkin, "Xen 0wning trilogy," *Invisible Things Lab*, 2008.

[50] C. mitre, "CVE-2007-4993 : Xen guest root can escape to domain 0 through pygrub," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4993.

[51] ——, "CVE-2008-2100 : VMware buffer overflows in VIX API let local users execute arbitrary code," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2100.

[52] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," 2005.

[53] A. M. D. Inc., "Amd64 virtualization codenamed "pacifica" technology : Secure virtual machine architecture reference manual," 2005.

[54] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel virtualization technology : Hardware support for efficient processor virtualization," 2006.

[55] A. M. D. Inc., "Amd-v nested paging," Whitepaper : http://developper.amd.com/assets/NPT-WP-11-final-TM.pdf, 2008.

[56] S. Jin, J. Ahn, S. Cha, and J. Huh, "Architectural support for secure virtualization under a vulnerable hypervisor," ser. MICRO-44, 2011.

[57] S. Jin and J. Huh, "Secure mmu : Architectural support for memory isolation among virtual machines," in *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, June 2011, pp. 217–222.

[58] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel virtualization technology : Hardware support for efficient processor virtualization." *Intel Technology Journal*, vol. 10, pp. 167 – 177, 2006.

[59] T. Alves and D. Felton, "Trustzone : Integrated hardware and software security," *ARM white paper*, vol. 3, no. 4, pp. 18–24, 2004.

[60] G. Kornaros, I. Christoforakis, O. Tomoutzoglou, D. Bakoyiannis, K. Vazakopoulou, M. Grammatikakis, and A. Papagrigoriou, "Hardware support for cost-effective system-level protection in multi-core socs," in *Digital System Design (DSD), 2015 Euromicro Conference on*. IEEE, 2015, pp. 41–48.

[61] J. Porquet, "Architecture de sécurité dynamique pour systèmes multiprocesseurs intégrés sur puce." Ph.D. dissertation, Université Pierre et Marie Curie (UPMC), 2010.

[62] J.Porquet, A.Greiner, and C.Schwarz, "Noc-mpu : A secure architecture for flexible co-hosting on shared memory mpsocs." in *Design, Automation Test in Europe Conference Exhibition*, ser. DATE'11, 2011.

[63] G. Plouviez, "Etude, spécification, vérification formelle de mécanismes de virtualisation sécurisés pour architecture many-cores." Ph.D. dissertation, Université Pierre et Marie Curie (UPMC), 2012.

[64] D. Champagne and R. B. Lee, "Scalable architectural support for trusted software," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, Jan 2010, pp. 1–12.

[65] R. C. Merkle, "Protocols for public key cryptosystems," in *Security and Privacy, 1980 IEEE Symposium on*. IEEE, 1980, pp. 122–122.

[66] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, "Nohype : Virtualized cloud infrastructure without the virtualization," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, 2010.

[67] K. Kortchinsky, "Hacking 3d (and breaking out of vmware)," *BlackHat USA*, 2009.

[68] R. Wojtczuk, "Subverting the xen hypervisor," *Black Hat USA*, vol. 2008, 2008.

[69] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA : ACM, 2007, pp. 494–505.

[70] ——, "A novel cache architecture with enhanced performance and security," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41. Washington, DC, USA : IEEE Computer Society, 2008, pp. 83–93.

[71] J. Szefer and R. B. Lee, "Architectural support for hypervisor-secure virtualization," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII, 2012.

[72] S. Jin, J. Ahn, J. Seol, S. Cha, J. Huh, and S. Maeng, "H-svm : Hardware-assisted secure virtual machines under a vulnerable hypervisor," *IEEE Transactions on Computers*, vol. 64, no. 10, pp. 2833–2846, 2015.

[73] N. Harris, D. Barrick, I. Cai, P. G. Croes, A. Johndro, B. Klingelhoets, S. Mann, N. Perera, and R. Taylor, "Lpar configuration and management working with ibm eserver iseries logical partitions," in *IBM Redbooks*, 2002.

[74] R.Buerki and A.-K. Rueegsegger, "Muen-an x86/64 separation kernel for high assurance." http://muen.codelabs.ch/muen-report.pdf.

[75] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4 : Formal verification of an os kernel," in *In Symposium on Operating Systems Principles*, ser. SOSP'09, 2009.

[76] Hitachi, "Hitachi embedded virtualization technology." [Online]. Available : http://www.hitachi-america.us/supportingdocs/forbus/ssg/pdfs/Hitachi_Datasheet_Virtage_3D_10-30-08.pdf

[77] G. H. Software, "Integrity multivisor." [Online]. Available : http://www.ghs.com/products/rtos/integrity_virtualization.html

[78] LPARBOX, http://lparbox.com/.

[79] Y. Dai, Y. Qi, J. Ren, Y. Shi, X. Wang, and X. Yu, "A lightweight vmm on many core for high performance computing," in *ACM SIGPLAN Notices*, vol. 48, no. 7. ACM, 2013, pp. 111–120.

[80] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[81] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13, 2013.

[82] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions." in *HASP@ ISCA*, 2013, p. 11.

[83] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." in *HASP@ ISCA*, 2013, p. 10.

[84] I. Corporation, "Intel Software Guard Extensions Programming Reference," https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[85] F. X. McKeen, C. V. Rozas, U. R. Savagaonkar, S. P. Johnson, V. Scarlata, M. A. Goldsmith, E. Brickell, J. T. Li, H. C. Herbert, P. Dewan *et al.*, "Method and apparatus to provide secure application execution," Jul. 21 2015, uS Patent 9,087,200.

[86] S. P. Johnson, U. R. Savagaonkar, V. R. Scarlata, F. X. McKeen, and C. V. Rozas, "Technique for supporting multiple secure enclaves," Mar. 3 2015, uS Patent 8,972,746.

[87] T. C. Group., "TPM main specification," https://trustedcomputinggroup.org/tpm-main-specification/.

[88] D. Grawrock, *Dynamics of a Trusted Platform : A building block approach*. Intel Press, 2009.

[89] LIP6, "VciBlockDevice Composant SoClib," http://www.soclib.fr/trac/dev/wiki/Component/VciBlockDevice.

[90] D. Gibson and B. Herrenschmidt, "Device trees everywhere," *OzLabs, IBM Linux Technology Center*, 2006.

[91] G. Likely and J. Boyer, "A symphony of flavours : Using the device tree to describe embedded hardware," in *Proceedings of the Linux Symposium*, vol. 2, 2008, pp. 27–37.

[92] L. Gaspar, "Crypto-processeur - architecture, programmation et évaluation de la sécurité." Ph.D. dissertation, Université Jean Monnet, 2012.

[93] M. J. Dworkin, "Sp 800-38a 2001 edition. recommendation for block cipher modes of operation : Methods and techniques," 2001.

[94] "Ieee standard for cryptographic protection of data on block-oriented storage media," *IEEE Std 1619-2007*, 2008.

[95] L.Martin, "Xts : A mode of aes for encrypting hard disks," *IEEE Security Privacy*, 2010.

[96] "Ieee standard for wide-block encryption for shared storage media," *IEEE Std 1619.2-2010*, 2011.

[97] D. Chakraborty, V. Hernandez-Jimenez, and P. Sarkar, "Another look at xcb," *Cryptography and Communications*, 2015.

[98] M. A. Simplicio, B. T. de Oliveira, C. B. Margi, P. S. Barreto, T. C. Carvalho, and M. Näslund, "Survey and comparison of message authentication solutions on wireless sensor networks," *Ad Hoc Networks*, vol. 11, no. 3, pp. 1221–1236, 2013.

[99] P. Rogaway, "Efficient instantiations of tweakable blockciphers and refinements to modes ocb and pmac," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2004, pp. 16–31.

[100] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Keccak sponge function family main document," *Submission to NIST (Round 2)*, vol. 3, p. 30, 2009.

[101] LIP6, "VciMwmrDma Composant SoClib," http://www.soclib.fr/trac/dev/wiki/Component/VciMwmrDma.

[102] N. I. of Standards and T. (NIST), "Recommendation for Password-Based Key Derivation," http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf.

[103] P. S. Kedar and V. Bhusari, "Using pbkdf2 pair & hybrid technique for authentication," *International Journal of Emerging Research in Management & Technology (ISSN)*, pp. 2278–9359.

[104] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off," in *Annual International Cryptology Conference*, 2003.

[105] P.Gauravaram, "Security analysis of salt||password hashes," in *2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*, 2012.

[106] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs : Characterization and methodological considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. New York : ACM Press, 1995, pp. 24–37.

[107] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *High Performance Computer Architecture, 2007. HPCA 2007.*

[108] LIP6, Lab-STICC, LabHC and CEA-LIST, "Dépôt SVN projet ANR TSUNAMY," https://www-soc.lip6.fr/svn/tsunamy.

[109] C. Dévigne, J. B. Bréjon, Q. Meunier, and F. Wajsbürt, "Executing secured virtual machines within a manycore architecture," in *2015 Nordic Circuits and Systems Conference (NORCAS) : NORCHIP International Symposium on System-on-Chip (SoC)*, 2015.

[110] C. Dévigne, J.-B. Bréjon, Q. L. Meunier, and F. Wajsbürt, "Executing secured virtual machines within a manycore architecture," *Microprocessors and Microsystems*, 2017, extended papers from the 2015 Nordic Circuits and Systems Conference.

Résumé

Les architectures manycore, qui comprennent un grand nombre de cœurs, sont un moyen de répondre à l'augmentation continue de la quantité de données numériques à traiter par les infrastructures proposant des services de *cloud computing*. Ces données, qui peuvent concerner des entreprises aussi bien que des particuliers, sont sensibles par nature, et c'est pourquoi la problématique d'isolation est primordiale. Or, depuis le début du développement du *cloud computing*, des techniques de virtualisation sont de plus en plus utilisées pour permettre à différents utilisateurs de partager physiquement les mêmes ressources matérielles. Cela est d'autant plus vrai pour les architectures manycore, et il revient donc en partie aux architectures de garantir la confidentialité et l'intégrité des données des logiciels s'exécutant sur la plateforme.

Nous proposons dans cette thèse un environnement de virtualisation sécurisé pour une architecture manycore. Notre mécanisme s'appuie sur des composants matériels et un logiciel hyperviseur pour isoler plusieurs systèmes d'exploitation s'exécutant sur la même architecture. L'hyperviseur est en charge de l'allocation des ressources pour les systèmes d'exploitation virtualisés, mais ne possède pas de droit d'accès sur les ressources allouées à ces systèmes. Ainsi, une faille de sécurité dans l'hyperviseur ne met pas en péril la confidentialité ou l'intégrité des données des systèmes virtualisés.

Notre solution est évaluée en utilisant un prototype virtuel précis au cycle et a été implémentée dans une architecture manycore à mémoire partagée cohérente. Nos évaluations portent sur le surcoût matériel et sur la dégradation en performance induits par nos mécanismes. Enfin, nous analysons la sécurité apportée par notre solution.

Mots-Clés : Virtualisation, Sécurité, Architecture Manycore, Hyperviseur, Isolation

Abstract

Manycore architectures, which comprise a lot of cores, are a way to answer the always growing demand for digital data processing, especially in a context of cloud computing infrastructures. These data, which can belong to companies as well as private individuals, are sensitive by nature, and this is why the isolation problematic is primordial. Yet, since the beginning of cloud computing, virtualization techniques are more and more used to allow different users to physically share the same hardware resources. This is all the more true for manycore architectures, and it partially comes down to the architectures to guarantee that data integrity and confidentiality are preserved for the software it executes.

We propose in this thesis a secured virtualization environment for a manycore architecture. Our mechanism relies on hardware components and a hypervisor software to isolate several operating systems running on the same architecture. The hypervisor is in charge of allocating resources for the virtualized operating systems, but does not have the right to access the resources allocated to these systems. Thus, a security flaw in the hypervisor does not imperil data confidentiality and integrity of the virtualized systems.

Our solution is evaluated on a cycle-accurate virtual prototype and has been implemented in a coherent shared memory manycore architecture. Our evaluations target the hardware and performance overheads added by our mechanisms. Finally, we analyze the security provided by our solution.

Mots-Clés : Virtualization, Security, Manycore Architecture, Hypervisor, Isolation