Université Pierre et Marie Curie

Mastère de sciences et technologies

MENTION INFORMATIQUE
2014 – 2015

Spécialité : SESI

SYSTÈMES ÉLECTRONIQUES ET SYSTÈMES INFORMATIQUES

Exécution de plusieurs systèmes d'exploitation sur une puce manycore CC-Numa sécurisée

RAPPORT DE RÉALISATION

02 septembre 2015

PRÉSENTÉ PAR

JEAN-BAPTISTE BRÉJON

ENCADRANT

QUENTIN MEUNIER

Laboratoire d'accueil : LIP6

SOC - ALSOC

Table des matières

Table des figures

Chapitre 1

Contexte du stage

Ce stage s'inscrit dans le cadre du projet ANR TSUNAMY [1]. Ce projet adresse la problématique de la manipulation sécurisée des données personnelles et privées sur l'architecture CC-NUMA [2] TSAR [1] [3]. Ce stage s'intitule *Exécution de plusieurs systèmes d'exploitation sur une puce manycore CC-Numa sécurisée*. Il vise à proposer une solution de confiance par construction permettant d'exécuter plusieurs systèmes d'exploitation (OS) de manière indépendante sur la même architecture, en garantissant le cloisonnement des systèmes d'exploitation entre eux.

1.1 L'architecture TSAR

TSAR est une architecture à mémoire distribuée organisée en clusters. Ceux-ci sont composés de :

— Quatre processeurs ayant chacun un cache de niveau 1 et partageant un cache de niveau 2.
— Un interconnect local
— Un concentrateur d'interruption (XICU)
— Un Direct Memory Access (DMA) [4]

Les clusters sont connectés entre eux par un réseau sur puce (NoC [5]) de topologie Mesh-2D.

1.2 La plateforme

La plateforme (figure 1.1) utilisée durant ce stage comporte quatre clusters. Le cluster 0 contient, en plus des composants déjà énoncés, des périphériques non répliqués :

— un contrôleur de disque (IOC)
— un contrôleur de terminaux (TTY)
— une Read Only Memory (ROM)

1. https ://www.tsunamy.fr/trac/tsunamy
2. Cache-Coherent Non Uniform Memory Access
3. Tera Scale Architecture : https ://www-soc.lip6.fr/trac/tsar
4. réalise des transfert de données de taille arbitraire d'un endroit de la mémoire à un autre
5. Network On Chip : permet le routage des requêtes en dehors des clusters

Chaque cluster contient un banc mémoire de capacité 62Mo (0x03e000000). La répartition des adresses par cluster est faite de la manière suivante :

Coordonnées (x, y) du cluster	Numérotation TSAR	Plage d'adresses
(0, 0)	0	0x00000000 – 0x03e00000
(0, 1)	1	0x40000000 – 0x43e00000
(1, 0)	2	0x80000000 – 0x83e00000
(1, 1)	3	0xc0000000 – 0xc3e00000

La plateforme TSUNAMY contient, devant chaque initiateur, une Hardware Address Translator (HAT) (voir section 1.6).

1.3 Système d'exploitation ALMOS

ALMOS [2] est un système d'exploitation développé au département SoC [6] du laboratoire LIP6 [7]. Il a été développé par Ghassan ALMALESS dans le cadre de sa thèse. ALMOS cherche à répondre aux problématiques liées aux architectures CC-NUMA manycore, notamment le placement optimisé des données.

Nous considérons plusieurs instances de système d'exploitation dans la plateforme qui seront toutes des instances d'ALMOS mais qui pourraient être un autre système d'exploitation. Les instances des OS seront présentes sur les disques de la plateforme au démarrage de la simulation.

1.4 Virtualisation

Le principe de la virtualisation [3] est de permettre à plusieurs OS de s'exécuter en même temps sur la même machine. Les motivations pour l'utilisation de cette approche sont nombreuses (utilisation optimale des ressources matérielles, faciliter la migration des machines virtuelles d'une machine physique à une autre...). Voici la définition de quelques termes qui seront utilisés tout au long de ce rapport :

— Une VM (Virtual Machine) : est une machine émulée par un hyperviseur, il s'agit d'un sous-ensemble de la plateforme TSAR complète.
— Un hyperviseur, aussi appelé moniteur de machines virtuelles, est un environnement d'exécution de VMs.
— Un système invité est un OS qui s'exécute dans une VM.
— Un système hôte est le OS dans lequel l'hyperviseur s'exécute.

Par l'intermédiaire de la virtualisation, chaque VM doit avoir l'impression de s'exécuter en mode privilégié, d'avoir un espace d'adressage continu et de manipuler de la mémoire physique. Elle doit aussi avoir accès à des périphériques, bien que ceux-ci ne sont pas forcement les mêmes que ceux de la machine hôte.

6. http ://www-soc.lip6.fr/
7. http ://www.lip6.fr/

Il existe trois approches pour l'accès au matériel : la full-virtualisation, la para-virtualisation et l'utilisation directe du matériel.

1.4.1 La full-virtualisation

Dans cette approche, le matériel est complètement émulé par l'hyperviseur. L'avantage est qu'il n'y a pas besoin de modifier le code du système invité, il est donc possible de virtualiser n'importe quel SE. Mais il y a aussi un inconvénient de taille lié aux performances. En effet, dès que le système invité veut accéder au matériel, il doit passer par l'hyperviseur. De plus ces requêtes ne sont pas forcement optimisées car le système invité n'a aucune connaissance du matériel réellement présent sur la machine hôte.

1.4.2 La para-virtualisation

Le principe de cette approche consiste à modifier le système invité de manière à ce qu'il fonctionne en collaboration avec l'hyperviseur. Le système invité associe une page vers une adresse physique sans passer par l'hyperviseur. Ainsi les accès aux périphériques s'effectueront plus rapidement.

1.4.3 Solution matérielle

Cette solution est apparue en 2003 avec Intel-Vt, et a été rajoutée dans AMD-V depuis 2006. Le principe est d'ajouter un niveau de privilège pour les VMs. Ainsi lors d'un appel système ou d'une instruction privilégiée exécutée par une VM, celle-ci peut directement exécuter son instruction sur le processeur qui appellera une fonction de l'hyperviseur. Avec cette solution on peut utiliser les instructions spéciales fournies par les processeurs sans avoir à modifier le système invité.

Ces trois solutions ne conviennent pas au projet ANR TSUNAMY, car elles donnent trop de pouvoir à l'hyperviseur. On souhaiterait en effet lui interdire l'accès à la mémoire des VMs une fois celles-ci lancées. Cela n'est possible que dans une architecture ou les ressources sont physiquement distinctes.

1.5 L'hyperviseur

Un hyperviseur [4], [5] permet la virtualisation d'un ou plusieurs systèmes d'exploitation sur la même plateforme.

On peut distinguer deux types d'hyperviseur. Pour le premier, qui est dit natif, le code s'exécute directement sur la plateforme. Il n'y a donc aucun intermédiaire entre le matériel et l'hyperviseur. Le second type d'hyperviseur est un logiciel qui s'exécute à l'intérieur d'un système d'exploitation. L'hyperviseur qui sera développé durant ce stage est du premier type.

le rôle de l'hyperviseur est de distribuer les ressources de la plateforme (clusters et canaux des périphériques) entre les différentes VMs virtualisées. Les clusters alloués pour les

VMs devront respecter une contrainte topologique, à savoir former une structure rectangulaire. Cette contrainte est justifiée par le fait qu'ALMOS doit avoir une connaissance de la topologie réelle pour être efficace.

L'hyperviseur doit se charger de démarrer et d'isoler les VMs.

Chaque VM sera exécutée non modifiée. Elle produira donc des adresses entre 0x0000 0000 et 0xFFFF FFFF. Or, on souhaite contraindre la VM aux clusters qui lui ont été alloués. La plateforme modifiée pour TSUNAMY fournit un composant, appelé HAT, réalisant la traduction des adresses émises par la VM (0x0000 0000 - 0xFFFF FFFF) vers l'espace mémoire des clusters qui ont été alloués à la VM (voir section 1.6). Les adresses contenues dans l'espace mémoire des clusters sont appelées adresses machine.

La HAT permet aussi de répondre à d'autres problématiques posées par le projet ANR TSUNAMY, notamment la manipulation privée des données et le fait que cette manipulation doit se faire de manière sécurisée. En effet l'hyperviseur peut être le sujet d'attaques ou de bugs potentiels, il ne doit donc pas pouvoir accéder à l'intérieur des machines virtuelles une fois celle-ci lancées (on le dit aveugle). Il en est de même pour les machines virtuelles, une fois lancées toutes les requêtes envoyées depuis l'intérieur de la machine virtuelle ne doivent pas pouvoir en sortir. La HAT doit néanmoins faire une exception lorsque la requête est à destination d'un périphérique non répliqué. Les HATs permettent aussi l'isolation des machines virtuelles vis-à-vis des autres machines virtuelles et de l'hyperviseur.

1.6 La HAT

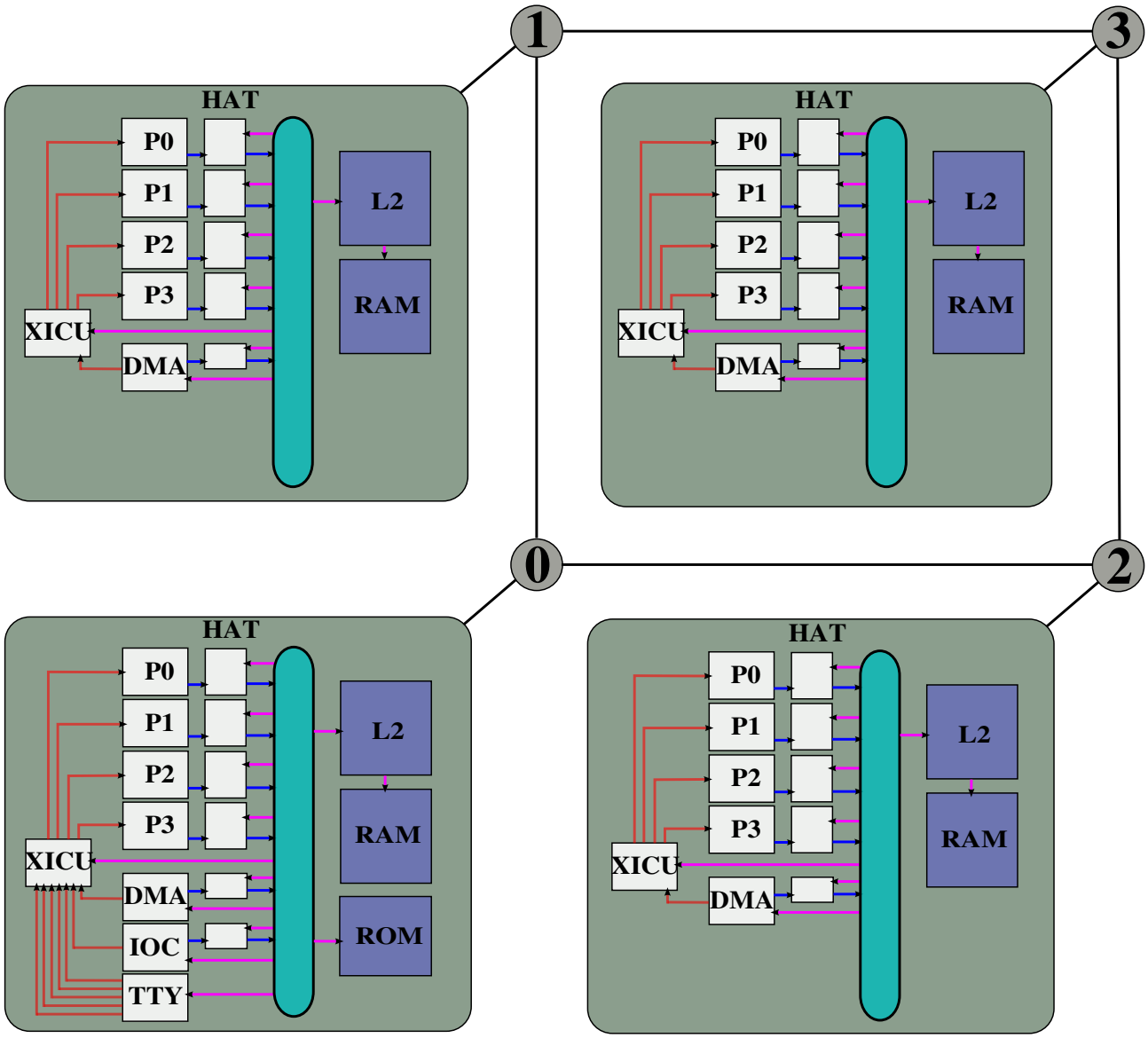

FIGURE 1.1 – Représentation de la plateforme

La figure 1.1 est une représentation de la plateforme telle qu'elle m'a été fournie. C'est une plateforme à 4 clusters basée sur l'architecture TSAR avec un nouveau composant devant chaque initiateur : la HAT.

La HAT est un composant matériel configurable placé devant tous les initiateurs [8] d'un cluster. De la même façon qu'une MMU [9], la HAT traduit à la volée des adresses physiques en adresses machine, mais à granularité cluster. La configuration de la HAT est confiée à l'hyperviseur, elle sera effectuée sur tous les clusters de la VM lors de son lancement. La HAT permet l'isolation des machines virtuelles entre elles et l'hyperviseur. Elle permet de restreindre la portée des requêtes émises par les machines virtuelles au(x) cluster(s) qui leurs ont été alloués.

Les registres à configurer dans les HATs sont les suivants :

HAT_MODE - R/W Ce registre permet de choisir le mode de la HAT parmi les 4 suivants :

HAT_MODE_BLOCKED Ne laisse passer aucune requête

HAT_MODE_FAILURE Erreur de traduction

HAT_MODE_IDENTITY Laisse passer les requêtes sans traduction

HAT_MODE_ACTIVATE Laisse passer les requêtes et effectue une traduction

HAT_MX0 - W : Contient la valeur en X de la position du cluster 0 de la machine virtuelle dans la plateforme

HAT_MY0 - W : Contient la valeur en Y de la position du cluster 0 de la machine virtuelle dans la plateforme

HAT_PXL - W : Contient le log2 de la largeur en X de la machine virtuelle

HAT_PYL - W : Contient le log2 de la largeur en Y de la machine virtuelle

HAT_TTY_PBA - W : Contient l'adresse physique de base du premier canal des TTYs

HAT_TTY_MASK - W : Permet de déterminer la plage d'adresse accessible

HAT_FB_PBA - W : Contient l'adresse physique de base du segment mémoire du Frame Buffer

HAT_FB_MASK - W : Permet de déterminer la plage d'adresse accessible

HAT_IOC_PBA - W : Contient l'adresse physique de base du canal du contrôleur de disque

HAT_IOC_MASK - W : Permet de déterminer la plage d'adresse accessible

La figure 1.2 montre un exemple de traduction d'adresse par la MMU et par la HAT dans une plateforme de 4 clusters. L'adresse est émise par un processeur se trouvant dans une VM lancée sur les clusters 1 et 3. La largeur en X de la VM est de 2 et de 1 en Y. Pxl représente le nombre de bits nécessaires pour coder la largeur en X de la plateforme, soit 1 bit dans l'exemple. Pyl est l'homologue de pxl mais pour la largeur en Y, soit 0 bit dans l'exemple.

La MMU traduit l'adresse 0x83681424 en 0xB1487424. Dans une VM de deux clusters, l'adresse commençant par 0xB est présente dans le banc mémoire du 2ème cluster (plage d'adresse physique : 0x8000 0000 - 0xFFFF FFFF). Ici, le 2ème cluster de la VM est le cluster 3, il correspond au 4ème cluster dans la plateforme (plage d'adresse machine : 0xC000 0000 - 0xFFFF FFFF).

8. composant ayant la capacité d'initier un transfert (couple requête réponse)

9. Memory Management Unit : fournit en s'indexant sur une table des traductions d'adresses à granularité page (une traduction physique pour chaque page virtuelle)

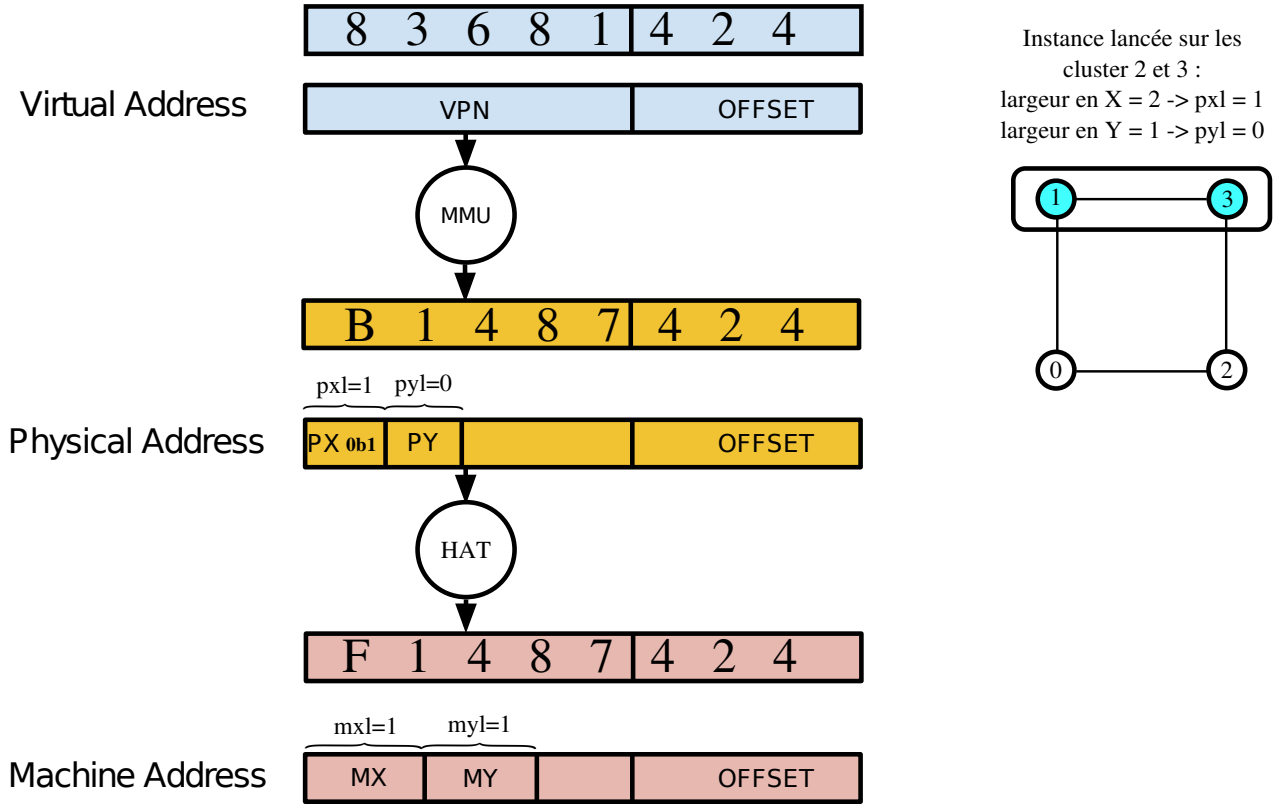

FIGURE 1.2 – Exemple de traduction

La HAT va donc fournir la traduction d'une adresse appartenant au 2ème cluster de la VM vers le 4ème cluster de la plateforme.

Tableau représentant les traductions générées par la HAT dans le cas de la VM présentée dans la figure 1.2 :

adresse émise	traduction		adresse émise	traduction
0x00000000	0x40000000		0x80000000	0xc0000000
0x10000000	0x50000000		0x90000000	0xd0000000
0x20000000	0x60000000		0xa0000000	0xe0000000
0x30000000	0x70000000		0xb0000000	0xf0000000
0x40000000	0x40000000		0xc0000000	0xc0000000
0x50000000	0x50000000		0xd0000000	0xd0000000
0x60000000	0x60000000		0xe0000000	0xe0000000
0x70000000	0x70000000		0xf0000000	0xf0000000

Chapitre 2

Définition et analyse du problème

Les objectifs de ce stage sont les suivants :

1. Adaptation d'une plateforme TSAR à 4 clusters pour la rendre compatible avec les besoins du projet (intégration des HATs, modification du routage des interruptions, adaptation du nombre de canaux des périphériques...).
2. Définition d'un mécanisme de boot de l'hyperviseur et des processeurs non actifs.
3. Écriture de la partie "VMLoader" (Virtual Machine Loader) de l'hyperviseur, en charge d'initialiser la mémoire du kernel, de configurer les HATs et de réveiller les processeurs.
4. Réalisation du monitoring des différents systèmes lancés via un multiplexage de leur terminaux.

2.1 Canaux des périphériques

Les canaux des périphériques ne posent de problème que dans le cas des périphériques non répliqués : le composant IOC et le composant TTY.

Composant MULTI_TTY :
Le composant TTY est déjà multi-canaux. Lors d'une utilisation classique, dans laquelle une instance d'ALMOS est lancée sur toute la plateforme, ce composant est initialisé avec 4 canaux car ALMOS a besoin de 4 terminaux. Comme présenté dans les objectifs, les TTYs devront être multiplexés entre les différentes VMs lancées. Le nombre de fenêtres "terminal" (processus Unix gérant la fenêtre) ne doit pas changer en fonction du nombre de VMs possibles sur la plateforme : une fenêtre pour le terminal hyperviseur, et 4 fenêtres pour toutes les instances d'ALMOS. L'affichage de ces 4 fenêtres devra donc pouvoir être changé pour afficher les informations de l'une ou l'autre des VMs lancées sur la plateforme.

Composant IOC :
Chaque VM a son propre disque, et la gestion de ces disques doit se faire de façon transparente. Or le composant actuel ne gère pas plusieurs disques.

Le composant IOC fait l'objet d'un autre stage se déroulant parallèlement à celui-là, ce composant devra à terme permettre de chiffrer et déchiffrer le code et les données d'une

VM. Néanmoins, pour pouvoir tester l'hyperviseur et lancer plusieurs VM, il sera nécessaire d'adapter le composant actuel de façon à le rendre multicanaux.

2.2 Routage des interruptions

Comme vu dans la figure 1.1, les interruptions des périphériques non répliqués (IOC, TTY) sont dirigées vers l'XICU du cluster 0 (cluster hyperviseur). Cela signifie qu'une VM lancée sur la plateforme n'a pas de moyen direct de se faire signifier la fin d'un transfert de l'IOC ou la frappe d'un caractère au clavier sur un des terminaux qui lui a été associé. Une des solution serait de passer par l'hyperviseur : l'hyperviseur signalerait à un des processeurs de l'instance la levée de l'interruption qui lui aura été signalée (via une IPI). Cela implique que l'XICU d'un cluster appartenant à l'instance puisse recevoir une requête de l'hyperviseur. Or, comme expliqué précédemment, on se l'interdit. Il faut donc développer un mécanisme pour transmettre l'interruption sans l'intervention directe de l'hyperviseur.

2.3 Démarrage d'une VM

Le démarrage d'une VM s'effectuera grâce à un shell lancé sur le terminal hyperviseur. Le but est de démarrer la VM dans un environnement sécurisé.

La séquence de démarrage d'une VM est la suivante :

1. Allocation des clusters.
2. Allocation des canaux des périphériques non répliqués.
3. Construction de la description de l'architecture.
4. Chargement depuis le disque du bootloader et du kernel du SE.
5. Isolation des clusters alloués à la VM.
6. Réveil des processeurs alloués à la VM par l'hyperviseur (envoi d'une IPI en spécifiant l'adresse du bootloader).

Le problème dans cette séquence de démarrage se situe dans les étapes 5 et 6. En effet, l'hyperviseur s'exécute sur le cluster 0 de la plateforme et ne peut pas envoyer d'IPI à une VM isolée. L'étape d'isolation des clusters consiste en la configuration des HATs de telle sorte qu'aucune écriture venant de l'extérieur de la VM, i.e. des clusters qui ne lui ont pas été alloués, ne soit pas acceptée par le cluster visé.

Inverser les étapes 5 et 6 n'est pas non plus satisfaisant, car cela signifierait que la VM démarre dans un environnement non isolé, et peut donc accéder à toute la plateforme. ALMOS a été compilé pour s'exécuter dans le cluster 0 de la VM à une certaine adresse : là ou l'hyperviseur l'a chargé en mémoire (étape 4). À ce moment-là, l'isolation n'est pas encore réalisée, et les HATs ne fournissent donc pas de traduction. Les requêtes des processeurs ne sont donc pas dirigées vers la RAM du cluster où se trouve le code d'ALMOS. Il est donc nécessaire de trouver une autre solution qui conserve l'ordre des étapes 5 et 6.

Chapitre 3

Principe de la solution envisagée

3.1 Solution au problème des canaux des périphériques

En ce qui concerne l'IOC, la solution est simple : il faut plusieurs canaux. Le nombre de canaux optimal est déterminé par le minimum entre le nombre de disques et le nombre de clusters - 1 ; cette valeur peut être vue comme le nombre maximum de VM qui peuvent être lancées sur la machine.

Pour les TTYs, il est nécessaire de créer un nouveau composant à partir de celui existant. Ce nouveau composant permettra à une fenêtre de supporter N "fichiers" (un fichier étant un terminal virtuel utilisé par une VM ALMOS). Le multiplexage de ces fichiers sur la fenêtre sera effectué par le nouveau composant, et le choix de la VM à afficher par une commande entrée par l'utilisateur sur le terminal hyperviseur. L'utilisateur pourra spécifier dans la commande le numéro du terminal virtuel que l'on souhaite afficher sur les 4 fenêtres des VMs ALMOS.

3.2 Solution au problème du routage des interruptions

La solution envisagée pour répondre à ce problème est d'utiliser un périphérique permettant d'émettre une requête sur le réseau lorsqu'il reçoit une interruption matérielle. Ce composant est appelé IOPIC pour "In/Out Programmable Interrupt Controller". et celui-ci est déjà présent dans la bibliothèque Soclib [1].

Ce nouveau périphérique sera placé dans le cluster où se trouve les périphériques non-répliqués, c'est à dire le cluster 0. Il permet de traduire un certain nombre d'interruptions matérielles (HWI) en interruptions logicielles (WTI). L'adresse à laquelle chaque WTI sera envoyée peut être configurée, l'hyperviseur y mettra l'adresse de l'XICU correspondant au cluster 0 d'une VM.

Ce sera donc l'IOPIC qui signifiera, via le réseau sur puce, à l'XICU du cluster 0 d'une VM qu'un transfert est terminé sur un des périphériques alloués à cette VM.

Les fenêtres peuvent afficher alternativement les informations des VMs. Il faut donc que les interruptions générées lorsqu'une touche est frappée soient redirigées vers la bonne VM.

1. http ://www.soclib.fr/trac/dev

L'IOPIC devra donc être reconfiguré lorsque l'utilisateur fera la demande d'afficher un autre terminal virtuel.

3.3 Démarrage d'une VM

L'idée pour le démarrage d'une VM est que chaque processeur contenu dans les clusters appartenant à la future VM exécute du code hyperviseur permettant l'isolation de la VM : ce code doit effectuer l'activation de la HAT du processeur qui l'exécute. Autrement dit, la future VM est isolée de l'intérieur.

L'hyperviseur devra donc réveiller un processeur appartenant à la VM en plaçant l'adresse du point d'entrée d'ALMOS dans une zone mémoire réservée aux arguments. Cet envoi d'IPI est possible car la VM n'est pas encore isolée. Lorsque le processeur se réveillera, il exécutera ce code (appelé RHA : Remote HAT Activation). Une fois dans la fonction RHA, le processeur se chargera de récupérer l'adresse du point d'entrée d'ALMOS préalablement stockée, d'activer sa HAT, puis de sauter au point d'entrée d'ALMOS.

Plus tard, ALMOS réveillera les autres processeurs, qui exécuteront la fonction RHA.

Chapitre 4

Identification des tâches à accomplir

4.1 Tâches

Hyperviseur
L'hyperviseur sera basé sur un code déjà existant. Les fonctionnalités déjà présentes sur celui-ci sont les suivantes :

— Fonction permettant la configuration du composant MULTI_TTY.
— Fonction permettant la configuration du composant IOC.
— API pour lire depuis le disque un fichier ELF (fichier exécutable d'Unix).

Les fonctions à rajouter sont les suivantes :

— Démarrage des processeurs inactifs et de l'hyperviseur.
— Construction de la structure représentant l'architecture.
— Configuration des HATs.
— Configuration de l'IOPIC.
— Développement du mécanisme de boot d'une VM.
— Développement des fonctions permettant la configuration du composant XICU.
— Développement du shell hyperviseur.

Matériel
En ce qui concerne le matériel, il faut créer deux autres composants :

MULTI_IOC : Ce composant est sensiblement le même que l'IOC mais supporte plusieurs disques et est multi-canaux.

MULTI_TTY_VT : comme pour l'IOC, le composant MULTI_TTY_VT sera fortement inspiré du composant MULTI_TTY mais supportera plusieurs fichiers (terminaux virtuels) par fenêtre (canal) et ajustera le nombre de canaux. Auparavant, le nombre de canaux était le même que le nombre de fenêtres, il faudra dans notre cas multiplier celui-ci par le nombre de terminaux virtuels.

4.1.1 Tâche 1 : Démarrage d'une VM sur le cluster 1

Cette tâche nécessitera l'implémentation des fonctions suivantes :

— Démarrage des processeurs inactifs et de l'hyperviseur.
— Construction de la structure représentant l'architecture.
— Configuration des HATs.
— Développement du mécanisme de démarrage d'une VM.
— Développement des fonctions permettant la configuration du composant XICU.

On redirigera le fil d'interruption de l'IOC et les quatre fils des TTYs vers le cluster 1 pour que ALMOS soit en mesure de lire ses applications depuis le disque.

4.1.2 Tâche 2 : Introduction IOPIC

Cette tâche consistera en l'introduction de l'IOPIC dans la plateforme ainsi que le développement des pilotes de configuration de celui-ci. Les lignes d'interruption de l'IOC et les 4 autres lignes d'interruption des TTYs n'ont plus besoin d'être redirigées dans la plateforme. Elles seront routées de façon logicielle. Cette partie nécessitera des modifications dans ALMOS pour que celui-ci supporte les WTI envoyées par l'IOPIC.

4.1.3 Tâche 3 : Composant MULTI_IOC - Démarrage de deux VM

La première VM sera placée sur les clusters 1 et 3 et la seconde sur le cluster 2. Cette tâche nécessitera l'implémentation du composant MULTI_IOC. La plateforme sera créée avec 9 terminaux physiques (fenêtres) qui seront distribués comme suit :
— les 4 premiers seront réservés à la première instance.
— les 4 suivants seront réservés à la seconde instance.
— le dernier terminal sera réservé à l'hyperviseur.

4.1.4 Tâche 4 : shell hyperviseur

Le développement du shell hyperviseur est nécessaire pour réaliser le multiplexage des terminaux, il doit dont être fait avant. La seule commande qui sera disponible sur le shell sera la suivante :

run : Permet le lancement d'une VM, l'utilisateur devra spécifier en argument de cette commande le numéro de l'instance du OS ainsi que le nombre de clusters sur lesquels devra être lancée la VM.

4.1.5 Tâche 5 : MULTI_TTY_VT

Cette tâche consistera en la conception du composant MULTI_TTY_VT, le développement des pilotes dans l'hyperviseur ainsi que l'introduction d'une nouvelle commande dans le shell hyperviseur :

switch : Permet de basculer l'affichage des quatre terminaux réservés aux VMs. L'utilisateur devra spécifier en argument de cette commande le numéro de la VM qu'il souhaite afficher sur les 4 terminaux.

Chapitre 5

Procédure de recette

La procédure de recette de ce stage sera principalement fonctionnelle. Elle consistera en la création et l'utilisation de deux instances ALMOS sur une plateforme décrite ci-dessous :

— 4 clusters
— Un composant MULTI_TTY_VT dans le cluster 0
— Un composant MULTI_IOC dans le cluster 0

Un cluster contient : 4 processeurs, une XICU, un DMA, une RAM, une HAT devant chaque initiateur.

La procédure de recette se déroulera comme suit :

1. Lancer une VM ALMOS sur le cluster 1 sans IOPIC
2. Lancer une VM ALMOS sur le cluster 1 avec IOPIC
3. Démarrage de deux VM ALMOS
4. Démarrage des VMs en utilisant le shell hyperviseur
5. Lancer l'application "hello" sur les deux VMs

Étape 1 : Validation de la construction de la structure décrivant l'architecture, de la configuration des HATs et du mécanisme de démarrage

Dans cette étape on s'attend à voir ALMOS démarrer sur les 4 terminaux qui lui ont été réservés. Le code du shell d'ALMOS se situe sur le disque. L'affichage du shell validera donc le bon fonctionnement du disque et que le routage de l'interruption est correct. Une fois le shell affiché, la saisie d'une touche au clavier ainsi que son affichage sur le shell d'ALMOS validera les TTYs ainsi que le routage des interruptions correspondant au 4 canaux associés à ALMOS.

Étape 2 : Validation de l'introduction de l'IOPIC ainsi que sa configuration

Le processus de validation de cette étape est le même que celui présenté à l'étape 1.

Étape 3 : Validation du fonctionnement du composant MULTI_IOC et de la bonne configuration de l'IOPIC pour les 2 VMs

On considèrera que le composant MULTI_IOC est validé si on observe que les ALMOS affichent leurs shells. Cela signifiera qu'ils ont réussi à lire le code du shell depuis leurs disques respectifs.

Étape 4 : Validation du bon fonctionnement du shell hyperviseur
On vérifiera que la commande "run" permet de lancer effectivement une VM sur le nombre de clusters souhaité et avec le disque souhaité, et que le placement correspond à celui voulu par l'hyperviseur.

Étape 5 : Réalisation du composant MULTI_TTY_VT
Validation du composant MULTI_TTY_VT, de la commande "switch" du shell hyperviseur ainsi que du re-routage des interruption lorsque celle ci est appelée. Cette étape sera validée si la commande "switch" permet effectivement de basculer l'affichage des 4 terminaux d'une VM à l'autre et que les interruptions sont bien redirigées vers la VM affichée sur les 4 terminaux. La redirection des interruptions nécessitera une reconfiguration de l'IOPIC. Cette étape sera considérée validée lorsque, après avoir basculé l'affichage des terminaux, l'appui d'une touche au clavier sera pris en compte par l'instance effectivement affichée à ce moment là.

Description de l'application "hello" : L'application "hello" d'ALMOS crée un thread sur chacun des processeurs de la VM. Dans lequel chacun des processeurs affiche un message contenant son numéro unique.

Chapitre 6

Réalisation

Dans ce chapitre nous présenterons les modifications qu'il a été nécessaire d'apporter aux composants afin de permettre à plusieurs VMs de pouvoir s'exécuter sur une plateforme TSAR modifiée, ainsi que le code de l'hyperviseur réalisant la procédure de démarrage d'une VM.

Les composants qu'il a été nécessaire de modifier sont les suivants :

— MULTI_TTY → MULTI_TTY_VT : introduction de canaux virtuels

— IOC → MULTI_IOC : modification pour le support de plusieurs canaux

Il a été nécessaire de développer un autre composant qui n'apparait pas dans la spécification. Il s'agit du composant MULTI_HAT, ce composant est nécessaire au bon fonctionnement des VMs, il permet de traduire les adresses physiques émises par le composant MULTI_IOC en adresses machines.

6.1 Composant MULTI_TTY_VT

Ce composant permet de permuter l'affichage d'un terminal (fenêtre XTerm) entre plusieurs terminaux virtuels. Il peut donc y avoir entre 1 et N terminaux, et 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.

Dans le cadre du projet Tsunamy ce composant est utilisé comme suit :

	Terminal 0	Terminal 1	Terminal 2	Terminal 3	Terminal 4
VT0	Hyperviseur	Vide	Vide	Vide	Vide
VT1	Vide	$Term_0\,VM_0$	$Term_1\,VM_0$	$Term_2\,VM_0$	$Term_3\,VM_0$
VT2	Vide	$Term_0\,VM_1$	$Term_1\,VM_1$	$Term_2\,VM_1$	$Term_3\,VM_1$
..	...	...	...	...	...
VTN	Vide	$Term_0\,VM_N$	$Term_1\,VM_N$	$Term_2\,VM_N$	$Term_3\,VM_N$

Le nombre de terminaux a été fixé à 5, un pour l'hyperviseur et 4 pour les VMs. Le nombre de terminaux virtuels correspond au nombre maximum de VMs pouvant être exécutées

simultanément sur la plateforme. Le nombre maximum de VMs devrait être le maximum entre le nombre nombre de cluster - 1 (le cluster 0 est reservé à l'hyperviseur) et le nombre d'images disque (contenant le code d'une instance d'un OS).

La présentation de ce composant se fera en 3 parties. Nous présenterons d'abord l'écriture dans un canal puis la gestion des interruptions et enfin le multiplexage des terminaux virtuels.

6.1.1 Écriture dans un canal

Le segment de l'espace d'adressage attribué à ce composant est organisé comme ci-dessous :

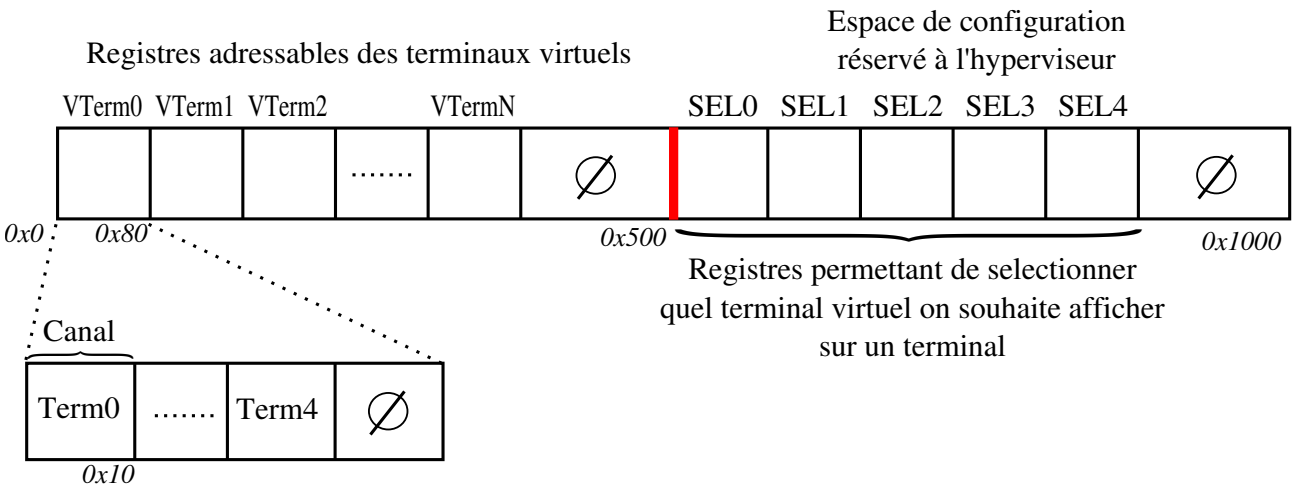

FIGURE 6.1 – Segment mémoire du composant MULTI_TTY_VT

Les canaux contenus dans le segment VTerm0 (terminal virtuel 0) sont réservés à l'hyperviseur, les autres segments VTerms seront attribués aux VMs. Chaque VTerm contient 5 canaux, à chaque canal est associé un fichier Unix dans lequel on stockera les informations écrites par les VMs dans ce canal. Les registres SELs permettent à l'hyperviseur de basculer l'affichage d'un terminal entre les différents VTerm.

La figure 6.2 représente la sélection du fichier Unix (émulant un buffer) Term1_1.txt créé par la simulation pour stocker les informations du terminal virtuel 1 de la VM 1 par le terminal 1. Dans le nom des fichiers Unix le premier numéro détermine sur quel terminal il seront affichés, le second numéro représente le numéro du terminal virtuel (numéro de VM). Le numéro '1' est le premier numéro des 2 fichiers représentés, ils seront donc multiplexés par la valeur du registre SEL1 pour être affichés sur le terminal 1. Le terminal affiche le fichier Term1_1.txt, cela signifie que ce sont les informations de la VM1 qui sont affichées sur ce terminal.

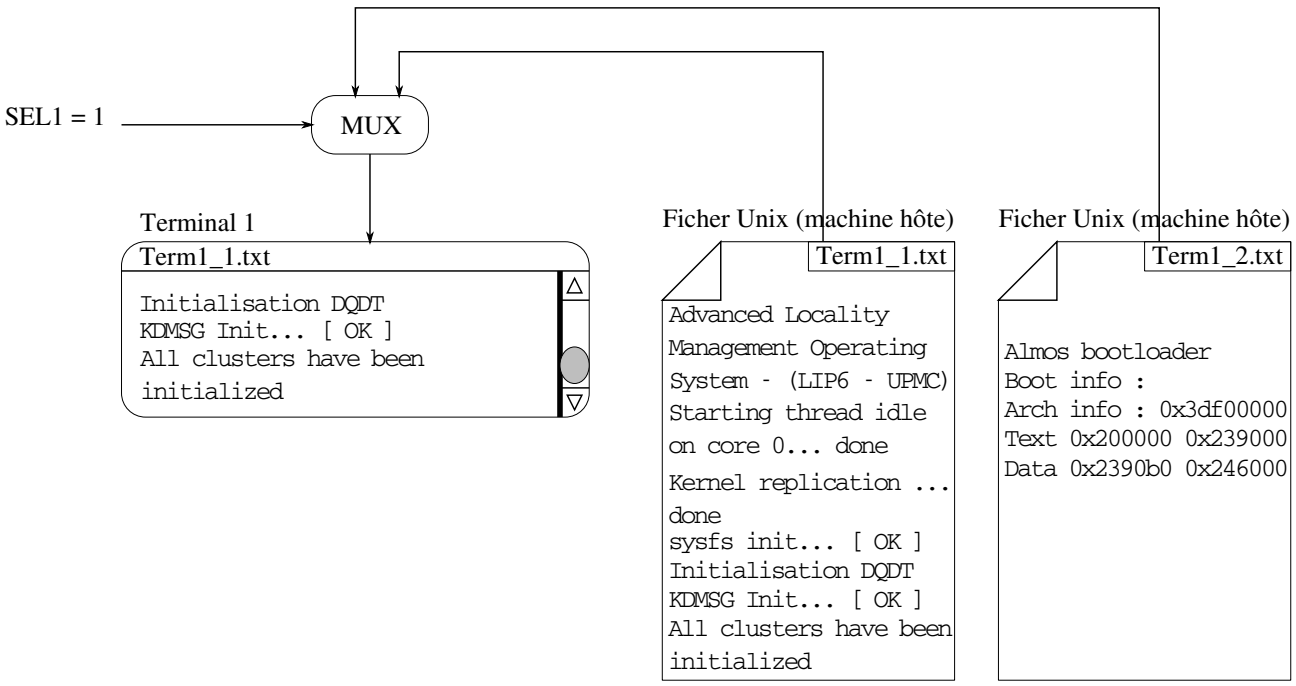

FIGURE 6.2 – Exemple de sélection de fichier par le composant MULTI_TTY_VT

6.1.2 Gestion des interruptions

Une interruption est générée à chaque appui sur une touche dans un terminal. Le composant MULTI_TTY_VT a donc autant de fils d'interruptions que de terminaux (fenêtres XTerm). Le problème était de pouvoir rediriger les interruptions vers les différentes instances (cf. section 3.2). Pour cela on utilise un composant, l'IOPIC (Input/Output Programmable interrupt controler), qui prend en entrée les interruptions du MULTI_TTY_VT et génère une IPI (Inter-Process Interrupt) en sortie lorsqu'une interruption est active. L'IPI envoyée est à destination du contrôleur d'interruption d'un cluster, celui-ci ayant été alloué à une VM. Comme son nom l'indique, l'IOPIC est un composant programmable, l'hyperviseur peut donc changer dynamiquement l'adresse du contrôleur d'interruption à laquelle sera envoyée l'IPI. Cela nous permet d'envoyer l'interruption à la VM dont l'affichage se trouve sur le terminal au moment où la touche est appuyée.

6.1.3 Multiplexage des terminaux virtuels

Le multiplexage des terminaux est réalisé grâce à un ensemble de registres, les registres SEL. Il y a un registre SEL par terminal (fenêtre Xterm). Dans le cadre du projet Tsunamy, ces registres sont à l'usage de l'hyperviseur. Les valeurs écrites dans ces registres sont des numéros de Vterm. Par exemple, l'écriture par l'hyperviseur de la valeur 2 dans le registre SEL1 entraîne l'affichage du Vterm2 dans le terminal 1.

La fonction de l'hyperviseur réalisant cette fonctionnalité est `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`.

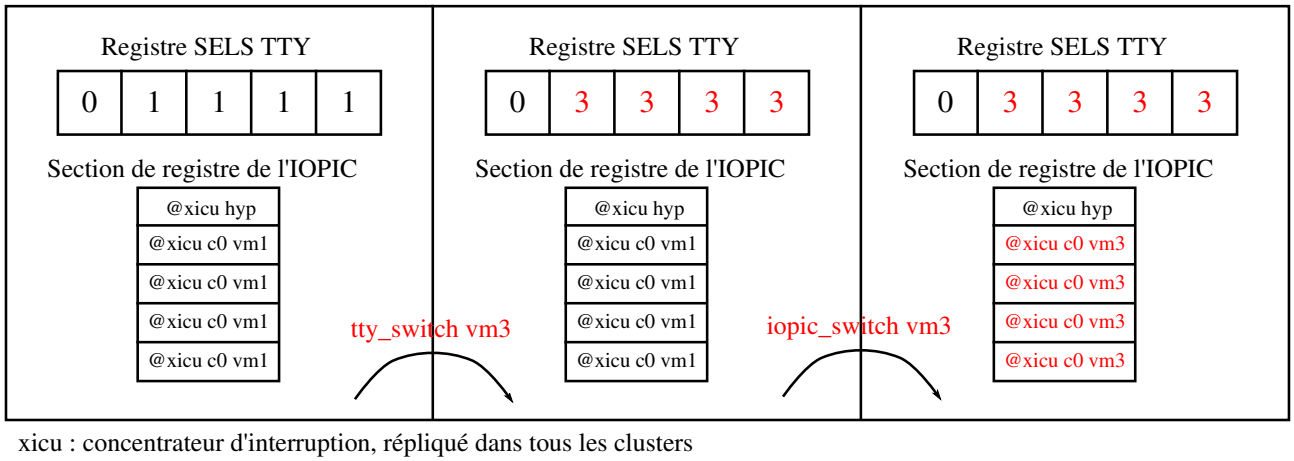

FIGURE 6.3 – Exécution de la fonction `switch_display`

La fonction `tty_switch` permet de modifier l'affichage des terminaux de la VM1 vers la VM3 (voir figure 6.3). La fonction `iopic_switch` permet de rediriger les interruptions des terminaux vers l'XICU, en charge de ces interruptions, de la VM3.

6.2 Composant MULTI_IOC

Ce composant est une adaptation "multi-canaux" du composant IOC déjà existant. Les modifications apportées à ce composant concernent non seulement les canaux, mais aussi la gestion de plusieurs disques. Un disque est associé à un canal, chaque VM se voit donc associer un disque.

Un problème est survenu lors de l'intégration du composant dans la plateforme. En effet, les registres adressables du composant MULTI_IOC (par canal) sont les suivants :

- W BLK_DEV_BUFFER_REG → adresse source/destination (dépend de l'opération read/write)
- W BLK_DEV_LBA_REG → Logical Block Address : numéro de bloc dans le disque
- W BLK_DEV_COUNT_REG → Taille en nombre de blocs
- W BLK_DEV_OP_REG → Opération : Read/Write
- R BLK_DEV_STATUS_REG → Fournit l'état (SUCCESS, ERROR, READY, BUSY) du canal et acquitte l'interruption
- W BLK_DEV_IRQ_ENABLE_REG → Booléen, autorisant la génération d'une interruption à la fin d'une transaction

Le registre qui pose problème est le registre `BLK_DEV_BUFFER_REG`. L'adresse fournie par l'OS est une adresse physique (voir figure 1.2). 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 une HAT. Avec la traduction d'une HAT les adresses émises par le disque seront redirigées vers les clusters de la VM qui a initiée la transaction.

Cependant le composant MULTI_IOC est multi-canaux. Un canal est affecté à une VM, il faut donc que la HAT devant le disque soit capable de fournir une traduction différente

pour chaque canal.

6.2.1 Composant MULTI_HAT

Pour répondre au problème posé par le composant MULTI_IOC, il a fallu développer un composant HAT multi-canaux nommé MULTI_HAT. Ce composant a le même nombre de canaux que le composant MULTI_IOC, et un ensemble de registres associé à chaque canal. Chaque ensemble de registres devra être configuré par l'hyperviseur lors du déploiement d'une VM. Il sera configuré de la même façon que les HATs, présentes devant les processeurs, allouées à la VM.

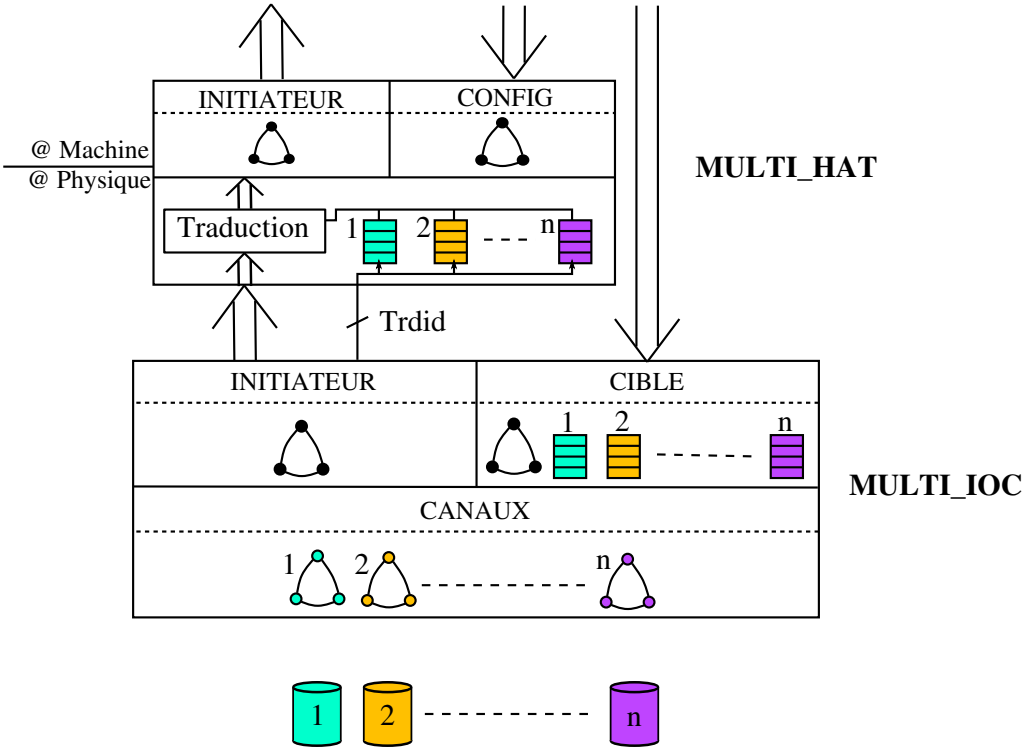

FIGURE 6.4 – Fonctionnement MULTI_HAT - MULTI_IOC

Comme expliqué dans la section 6.1.3 l'adresse écrite par l'OS dans le registre "BUFFER" du composant MULTI_IOC est une adresse physique. Elle doit donc être traduite par le composant MULTI_HAT pour que les données lues/écrites le soient depuis/vers le bon cluster. Un autre problème est apparu du coté du composant MULTI_HAT : comment faire pour que le composant MULTI_HAT sache quel ensemble de registres utiliser pour traduire l'adresse ?

Par exemple, sur la figure 6.4, la VM1 utilise le canal numéro 1 pour effectuer une transaction avec le composant MULTI_IOC. Lorsque l'automate initiateur envoie une requête, il faut que le composant MULTI_HAT traduise l'adresse de destination de cette requête en utilisant l'ensemble de registres numéro 1. Or, cette information n'existe pas dans la requête

envoyée par le composant MULTI_IOC. Néanmoins un champ de cette requête n'était jamais utilisé par le composant MULTI_IOC, il s'agit du champ TRDID (utilisé pour réaliser la cohérence des caches). Le composant MULTI_IOC utilise donc ce champ pour transmettre le numéro du canal. Le composant MULTI_HAT utilise ce numéro de canal pour choisir quel ensemble de registres utiliser afin d'effectuer la bonne traduction.

6.3 Démarrage d'une VM

L'idée de cette tâche était de faire exécuter du code de l'hyperviseur aux processeurs sur lesquels on souhaitait déployer la nouvelle VM (ce code est en fait une fonction nommée `remote_hat_activation` : RHA). Cette fonction devait activer la HAT du processeur qui l'exécutait puis faire sauter le processeur au point d'entrée de l'OS.

6.3.1 Copie du code RHA

Le code RHA est situé (comme tout le code hyperviseur) dans la ROM, donc dans le cluster 0 de la plateforme. Après l'activation de sa HAT par le processeur, l'espace d'adressage accessible par ce processeur est réduit aux clusters qui constitueront la nouvelle VM, il n'a donc plus accès à la partie du code RHA qui le fait sauter au point d'entrée de l'OS.

Après avoir exécuté l'instruction d'activation des HATs, il faut que le processeur exécute l'instruction suivante dans le code source. Or comme la traduction est désormais active, l'adresse de la prochaine instruction est traduite par la HAT, il faut donc préalablement copier le code RHA à l'adresse produite par la HAT. Le même problème se pose lors de l'activation d'une MMU : la façon de le résoudre est d'écrire dans l'entrée de la table des pages correspondant à l'adresse physique actuelle la valeur de cette adresse ; c'est à dire créer une projection dans l'espace d'adressage virtuel identique à l'espace d'adressage physique ("mapping" identité). Les HATs fournissant des traductions à la granularité cluster, il faut copier le code RHA **au même décalage** dans le cluster hyperviseur et dans le premier cluster de la nouvelle VM. Or le code RHA se trouvant dans la ROM du cluster hyperviseur et le premier cluster de la nouvelle VM (c0-VM) ne contenant pas de ROM, il n'est pas possible de copier le code RHA au même décalage dans le cluster c0-VM. Néanmoins, les traductions sont possibles de RAM à RAM, ainsi le code RHA est dans la RAM de l'hyperviseur et dans la RAM premier cluster c0-VM au même décalage.

ALMOS, l'OS utilisé dans le cadre de mon stage, réserve une plage d'adresse pour le code du bootloader (point d'entrée de l'OS). L'hyperviseur va donc copier le code RHA depuis la ROM à cette plage d'adresse dans la RAM du cluster hyperviseur et dans la RAM du cluster c0-VM.

La figure 6.5 présente la procédure de démarrage incluant le placement du code RHA.

Sur la figure 6.5, le cluster de gauche représente le cluster hyperviseur. Tout le code l'hyperviseur est contenu dans la ROM. La section contenant les instructions correspond au code RHA.

1 - Dans un premier temps l'hyperviseur copie le code RHA dans la RAM du cluster 0 et dans la RAM du premier cluster de la VM (c0-VM) en création. Le décalage par rapport au

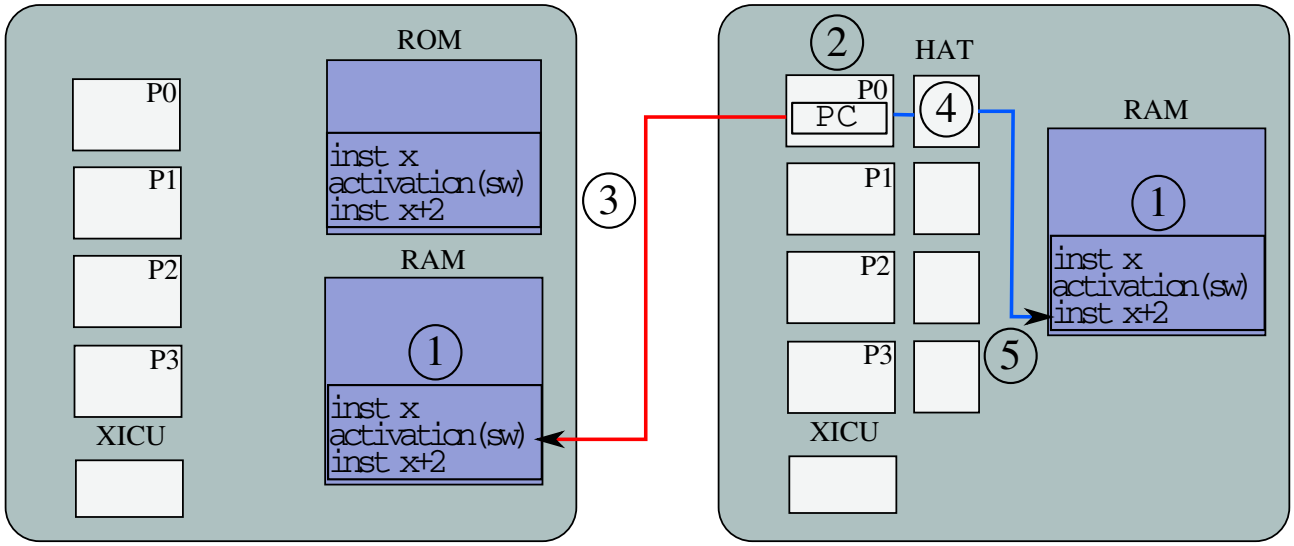

FIGURE 6.5 – Procédure de démarrage : placement RHA

début de la RAM doit être le même dans les 2 clusters, c'est la condition pour qu'une fois la HAT activée, le processeur se retrouve au même endroit dans le code.

2 - Le processeur 0 du cluster hyperviseur réveille le processeur du cluster c0-VM en lui spécifiant l'adresse de début du code RHA.

3 - Le processeur du cluster c0-VM exécute, depuis le cluster hyperviseur, l'instruction réalisant l'activation de sa HAT.

4 - La HAT est activée.

5 - La prochaine demande d'instruction de ce processeur est redirigée vers le cluster c0-VM dans la copie du code RHA. Il exécute donc le reste du code RHA dans le cluster c0-VM.

6.3.2 Appel de fonction

Le code RHA effectue des appels de fonction dans le code de l'hyperviseur qui se trouve dans la ROM. C'est un problème maintenant que le code RHA est exécuté depuis l'adresse où le compilateur l'a placé.

Les appels de fonction sont traduits par le compilateur en instruction `JAL : Jump And Link` (langage d'assemblage MIPS). Le format de cette instruction est le suivant :

— 6 bits sont réservés pour l'opcode (Operation Code : permet au processeur de savoir de quelle instruction il s'agit), en l'occurence il s'agit de l'instruction `JAL`.

— 26 bits qui serviront à déterminer l'adresse de destination du saut.

On voit que cette instruction ne fournit que 26 bits pour l'adresse de destination du saut, or le processeur a besoin de 32 bits pour effectuer le saut. Le processeur comble donc les 6 bits restant en utilisant les 6 bits de poids fort de l'adresse de l'instruction actuellement exécutée. Cela signifie que si le code est copié puis exécuté à une autre adresse, avec les 6 bits de poids fort différents, que celle pour laquelle il a été compilé, les valeurs immédiates contenues dans les instructions `JAL` ne correspondent plus à l'adresse de la fonction que l'on souhaite appeler.

Pour résoudre ce problème, le code source de la fonction RHA a été modifié pour que le compilateur ne génère plus des instructions `JAL` mais des instructions `JALR`. L'instruction `JALR` fait appel à un registre pour stocker l'adresse de saut. Elle permet donc de stocker une adresse de 32 bits et d'effectuer un saut direct à la bonne adresse, indépendamment de l'adresse où le code est exécuté.

Pour forcer le compilateur à générer des instructions `JALR`, les fonctions sont appelées via des pointeurs de fonction déclarés `volatile`. Le mot clé `volatile` (associé à une variable) spécifie au compilateur que cette variable ne doit pas être optimisée. À chaque appel de fonction le compilateur effectue des lectures pour récupérer l'adresse de cette fonction, cette adresse est alors stockée dans un registre et le compilateur utilise l'instruction `JALR` pour réaliser l'appel de fonction.

6.3.3 Cohérence des caches

Dans TSAR, le cache de niveaux 2 (L2) centralise la cohérence des cache de niveaux 1 (L1). Un L2 a la responsabilité de toutes les adresses de la RAM qui lui sont associées (RAM liée directement au cluster du L2). Un L2 doit garantir la cohérence des valeurs de "ses" lignes de cache contenues dans les caches L1. Or lorsqu'une VM est déployée (HAT activée) les L2 et les L1 de cette VM ne sont plus dans le même espace, il faut donc que les HATs devant les processeurs traduisent les requêtes de cohérence L2 vers L1 (requêtes dites P2M : CLEANUP[1]) ainsi que L1 vers L2 (requêtes dites M2P : INVAL[2], UPDATE[3]). Les HATS devant les processeurs traduisent donc les requêtes de cohérence.

Au démarrage de la machine tous les processeurs exécutent le même code. Ce code se situe dans la ROM, dans le cluster hyperviseur ; c'est donc le cache L2 du cluster hyperviseur qui est en charge de la cohérence des lignes de cache correspondant à ce code. Les lignes de cache correspondant au code exécuté se retrouvent dans les caches L1 de ces processeurs. Cela pose un problème après l'activation des HATs. En effet une fois les HATs activées, les processeurs sont susceptibles d'émettre des requêtes `CLEANUP` sur les lignes de cache relatives au code exécuté avant l'activation des HATs. Ces requêtes seront redirigées par les HATs vers un cache L2 se situant à l'intérieur de la VM, or les lignes de caches correspondant à ce code ne sont connues uniquement du cache L2 du cluster hyperviseur.

La figure 6.6 représente deux exemples d'exécution dans une plateforme 2 clusters.

Le premier exemple montre le cas où le cache L1 du cluster 1 émet une requête de cohérence (`CLEANUP`) de la ligne de cache 'A' avant l'activation de sa HAT (1, 2', 3').

— (1) Tous les caches exécutent le même code et chargent donc ce code dans leurs caches.
— (2') Le cache L1 émet une requête de cohérence qui est envoyée au cache L2 du cluster 0.
— (3') Le cache L2 trouve 'A' dans son cache et met à jour ses données de cohérence sur la ligne de cache 'A'.

1. CLEANUP : Invalidation spontanée du processeur ; signifie au L2 que ce L1 n'a plus l'adresse, ou une réponse à une requête INVAL d'un cache L2
2. INVAL : demande d'invalidation d'une ligne de cache du L2 à un L1
3. UPDATE : envoie de la nouvelle valeur d'une ligne de cache à un L1

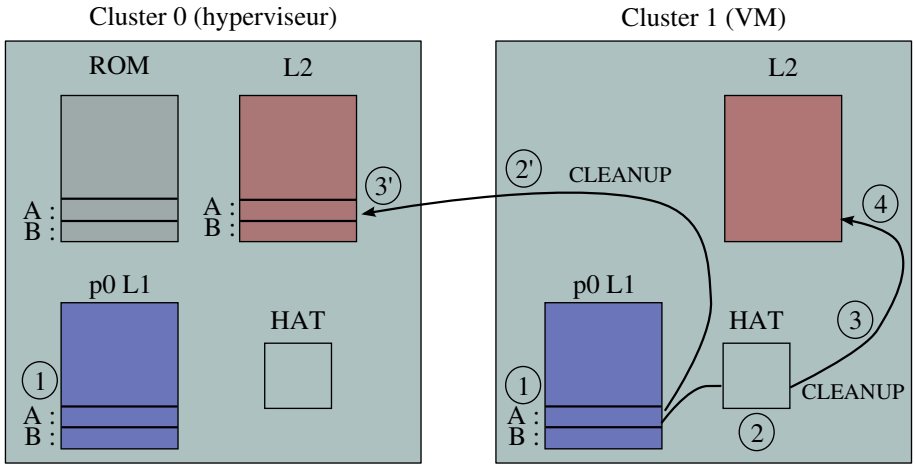

FIGURE 6.6 – Exemples d'exécution, mise en évidence du problème relatif à la cohérence des caches

Le second exemple montre le cas problématique où ce même cache L1 émet une requête de cohérence (`CLEANUP`) de la ligne de cache 'A' après l'activation de sa HAT (1, 2, 3 ,4).

— (1) Tous les caches exécutent le même code et chargent donc ce code dans leurs caches.
— (2) Activation de la HAT devant le processeur 0 du cluster 1.
— (3) Le cache L1 émet une requête de cohérence qui est traduite par la HAT et envoyée au cache L2 du cluster 1.
— (4) Le cache L2 du cluster 1 ne connait pas cette ligne ; la simulation s'arrête.

Pour remédier à ce problème, le processeur souhaitant activer sa HAT doit :

— Désactiver son cache.
— Vider le cache d'instructions et de données (forcer le cache à émettre des requêtes de cohérence pour toutes les adresses qu'il contient).
— Activer sa HAT.
— Sauter au point d'entrée de l'OS.
— Réactiver son cache dans le `delayed slot` [4] du saut.

4. Instruction toujours exécutée après une instruction de saut, caractéristique induite par le pipeline du processeur MIPS

Chapitre 7

Expérimentations et résultats

Dans ce chapitre nous présenterons les expérimentations effectuées dans le but de valider la procédure de recette. Nous verrons ensuite en détail un exemple d'exécution sur la plateforme finale.

7.1 Validation de la procédure de recette

La plateforme sur laquelle sera réalisée la validation de la procédure de recette est une plateforme à 4 clusters (2x2). le cluster 0 est réservé à l'hyperviseur, les clusters 1, 2 et 3 seront répartis sur 2 instances d'OS.

La procédure de recette présentée dans la spécification se découpe en 5 étapes :

1. Validation de la structure décrivant l'architecture et de la configuration des HATs
2. Validation de l'IOPIC
3. Validation du composant MULTI_IOC
4. Validation du shell hyperviseur
5. Validation du composant MULTI_TTY_VT

Étape 1 : validée

Cette étape consiste au lancement d'une VM d'ALMOS sur le cluster 1 sans l'IOPIC. Elle a permis de valider la bonne configuration des HATs par l'hyperviseur, la construction de la structure décrivant l'architecture ainsi que le mécanisme de démarrage d'une VM. La plateforme utilisée lors de cette étape est décrite sur la figure 1.1, à une différence près : les interruptions venant des TTYs et du composant IOC ont été redirigées vers le cluster 1.

Étape 2 : validée

L'étape 2 consiste au lancement d'une VM d'ALMOS sur le cluster 1 avec l'IOPIC. Elle a permis de valider la bonne introduction du composant IOPIC dans la plateforme ainsi que la configuration de ce composant. La plateforme utilisée diffère de celle de l'étape 1 par la présence de l'IOPIC dans le cluster 0 ainsi que par la redirection des interruptions vers ce composant.

Étape 3 : validée

Cette étape consiste au lancement de deux VMs en exploitant le composant MULTI_IOC. On a pu vérifier que les VMs affichaient bien leurs shell, ce qui signifie qu'elles ont réussi

à lire cette application depuis leurs disques respectifs. Cela a donc permis de valider ce composant.

Étape 4 : validée

L'étape 4 consiste au lancement d'une VM d'ALMOS en utilisant la commande `run` du shell de l'hyperviseur. On a pu vérifier que l'exécution de cette commande lançait bien une VM sur le nombre de clusters souhaité et avec le disque souhaité. Ces deux informations ont été passées en argument à la commande `run`.

Étape 5 : validée

La dernière étape consiste au lancement de deux VMs sur la plateforme en utilisant le composant MULTI_TTY_VT ainsi qu'à l'exécution de la commande `switch` du shell hyperviseur. On a pu observer que cette commande permettait bien de basculer l'affichage des terminaux et que la redirection des interruptions des TTYs vers la VM affichée sur les terminaux s'effectuait correctement.

7.2 Exécution de 2 VMs sur la plateforme finale

Nous allons maintenant voir un exemple d'exécution de 2 VMs sur la plateforme finale où l'on pourra observer la validation de chacune des étapes de la procédure de recette.

7.2.1 État initial

On voit sur la figure 7.1 les 5 terminaux de la simulation. Les terminaux 0, 1, 2 et 3 sont réservés aux VMs et le terminal 4 est utilisé par l'hyperviseur. Sur le terminal 4, la commande `run 1 0` a été rentrée au clavier. Cela signifie que l'utilisateur souhaite lancer la VM se trouvant sur le disque 0 sur 1 cluster.

7.2.2 Lancement de la première instance

On observe, sur la figure 7.2, le résultat de la commande `run 1 0`. On remarque que les terminaux 0, 1, 2 et 3 ont changé de nom : ils s'appellent maintenant "termX_1.txt" avec X allant de 0 à 3. Cela signifie que les terminaux 0, 1, 2 et 3 affichent les terminaux virtuels 0, 1, 2 et 3 de l'instance numéro 1 : en effet, lors de la création d'une instance, l'hyperviseur opère automatiquement l'équivalent de la commande `switch` (cf. section 6.1.3).

ALMOS utilise le terminal 0 pour afficher les informations du kernel. Le fait qu'il y ait des informations sur ce terminal signifie que cette VM a bien démarré et donc que les HATs ont été configurées correctement (**Validation de l'étape 1 1/2**).

On voit sur ce terminal la phrase suivante : "`Initialization of 1 Online Clusters`", comme il l'a été spécifié dans la commande, cette instance démarre bien sur 1 cluster. On valide donc la bonne construction de la représentation de l'architecture par l'hyperviseur (**Validation de l'étape 1 2/2**).

On peut déterminer l'endroit où a été placée l'instance en regardant les informations présentes sur le terminal 0. Plus précisément en regardant les lignes contenant les champs hw_id (hardware identifier), chaque ligne est affichée par un processeur différent. Cet identifiant est unique à chaque processeur. C'est une concaténation de la position en X et Y du

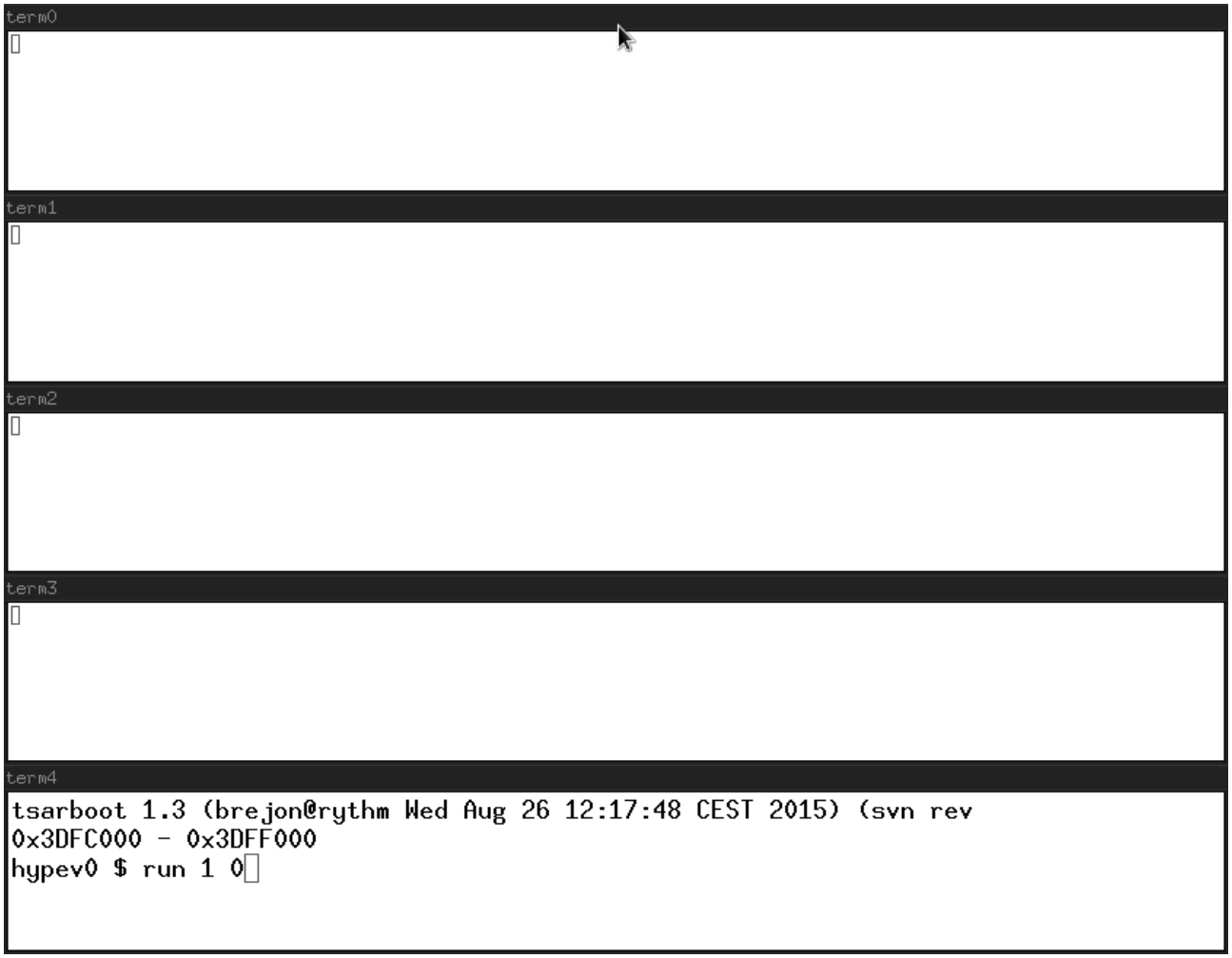

FIGURE 7.1 – État initial

cluster dans lequel il se trouve ainsi que son identifiant local dans ce cluster. L'identifiant local est codé sur 4 bits, la position en X sur 1 bit et la position en Y sur 1 bit.

Les champs hw_id sont interprétés comme cela :

— 16 → 0b010000 → x = 0, y = 1, identifiant local = 0
— 17 → 0b010001 → x = 0, y = 1, identifiant local = 1
— 18 → 0b010010 → x = 0, y = 1, identifiant local = 2
— 19 → 0b010011 → x = 0, y = 1, identifiant local = 3

L'instance a donc été lancée sur le cluster [x = 0, y = 1], l'hyperviseur étant lancé sur le cluster [x = 0, y = 0], il reste 2 clusters contigus libres (les clusters [x = 1, y = 0] et [x = 1, y = 1]) où l'on pourra lancer une instance de l'OS se trouvant sur le disque 1. La commande `run 1 0` a donc effectué le démarrage d'une VM (**Validation de l'étape 4**).

Sur le terminal 1 de la figure 7.2 on voit que le shell d'ALMOS s'affiche et que la commande `exec hello` a été tapée par l'utilisateur. Le fait que l'on puisse observer le shell d'ALMOS signifie que ALMOS a réussi à lire l'application `shell` depuis son disque (**Validation de l'étape 3**). L'observation de l'affichage de la commande tapée par l'utilisateur signifie

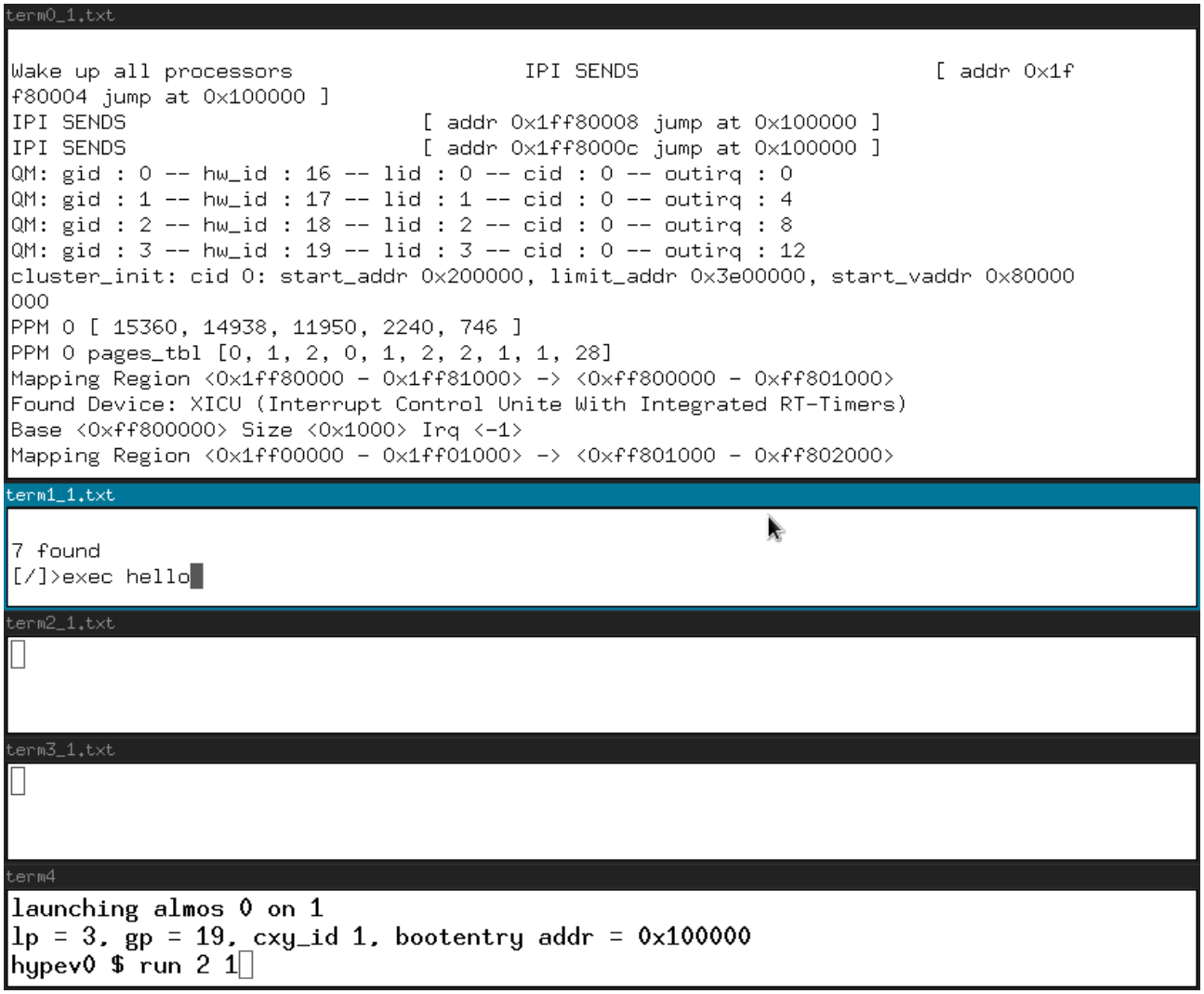

FIGURE 7.2 – Lancement de la première instance

que, lors du changement d'affichage effectué lors de la création de l'instance par l'hyperviseur, le composant IOPIC a bien été configuré (**Validation de l'étape 2**).

7.2.3 Lancement de la seconde instance

La figure 7.3 représente le résultat de la commande `run 2 1`. On peut voir que, comme pour le lancement de la première instance, le nom des terminaux 0, 1, 2 et 3 a changé : les terminaux s'appellent maintenant "termX_2.txt" avec X allant de 0 à 3. La phrase "`Initialization of 2 Online Clusters`" signifie que cette instance se lance bien sur 2 clusters.

Les champs hw_id sont interprétés comme cela :

— 32 → 0b100000 → x = 1, y = 0, identifiant local = 0

— 33 → 0b100001 → x = 1, y = 0, identifiant local = 1

— 34 → 0b100010 → x = 1, y = 0, identifiant local = 2
— 35 → 0b100011 → x = 1, y = 0, identifiant local = 3
— 48 → 0b110000 → x = 1, y = 1, identifiant local = 0
— 49 → 0b110001 → x = 1, y = 1, identifiant local = 1
— 50 → 0b110010 → x = 1, y = 1, identifiant local = 2
— 51 → 0b110011 → x = 1, y = 1, identifiant local = 3

On voit donc que l'instance est lancée, comme prévu, sur les clusters [x = 1, y = 0] et [x = 1, y = 1]. La présence de la commande `exec hello`, tapée par l'utilisateur, constate de la bonne re-configuration du composant IOPIC.

7.2.4 Commande switch

On voit, sur la figure 7.4, que la commande `switch_all 1` a été exécutée sur le terminal hyperviseur (term 4 sur la figure). La commande `switch_all` permet de basculer l'affichage des terminaux 0, 1, 2 et 3 en spécifiant en argument quels terminaux virtuels on souhaite afficher sur chacun des terminaux. On observe que, suite à l'exécution de cette commande, les terminaux 0, 1, 2 et 3 affichent maintenant les fichiers "termX_1.txt" avec X allant de 0 à 3.

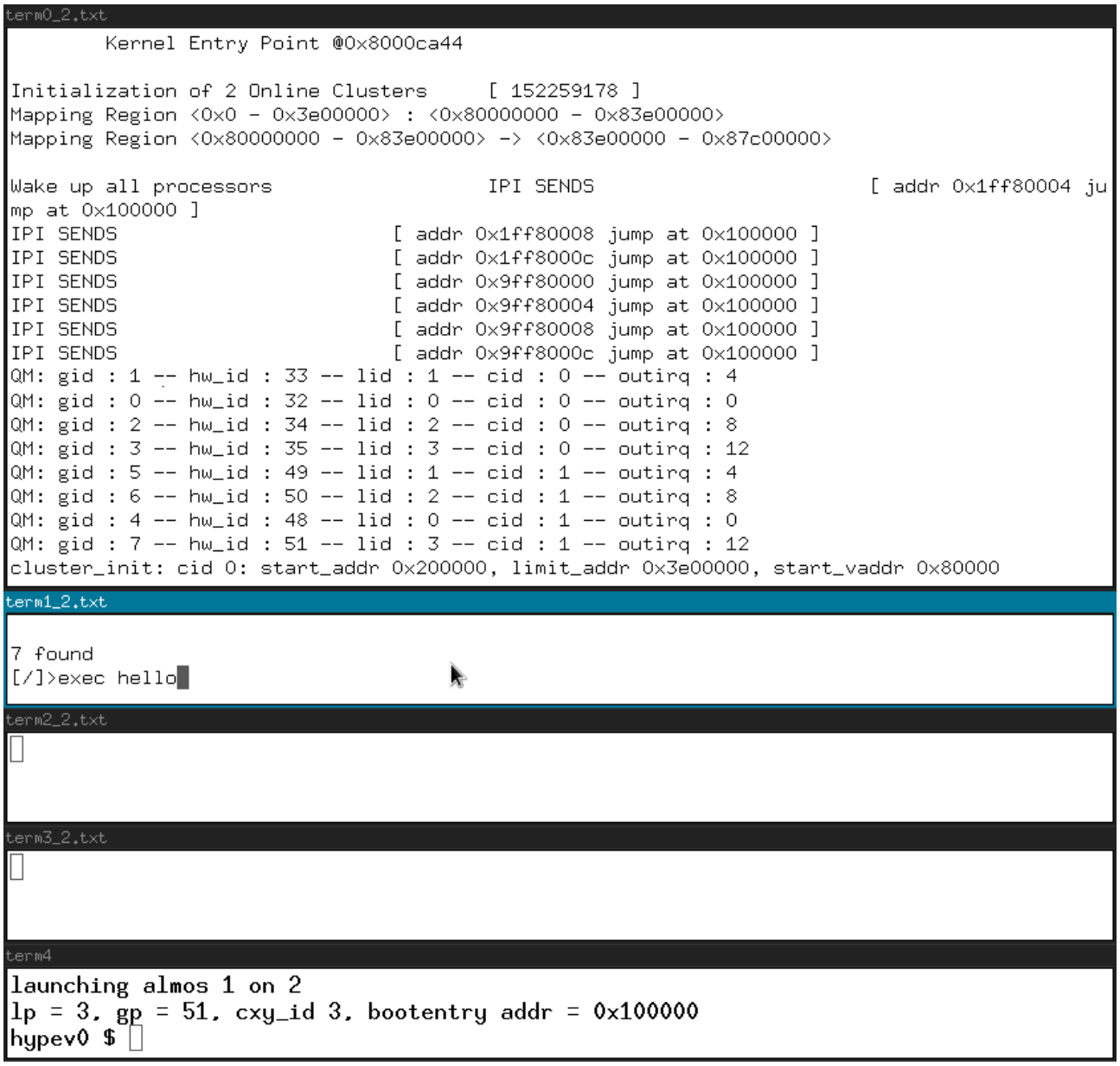

FIGURE 7.3 – Lancement de la seconde instance

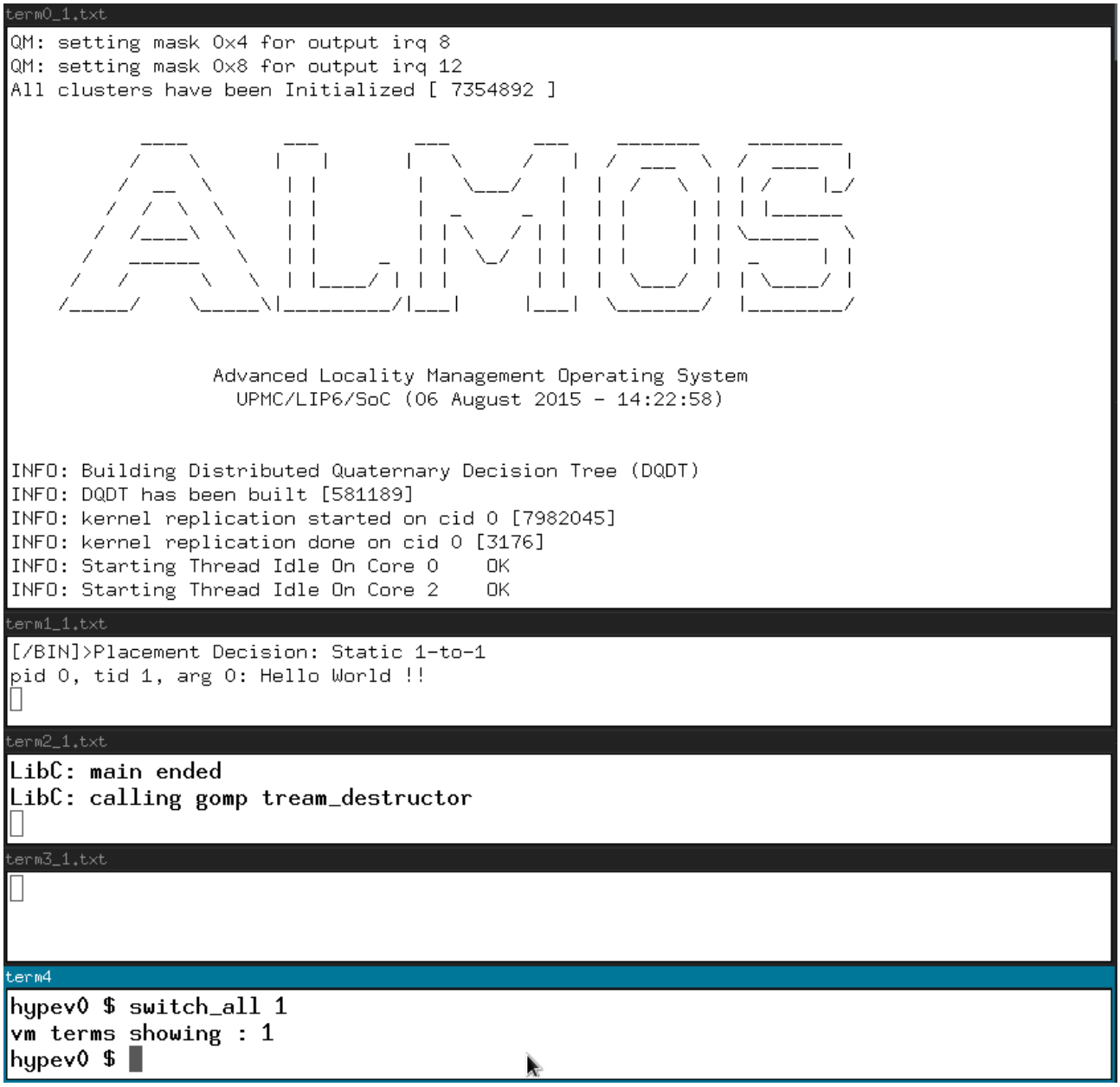

FIGURE 7.4 – Résultat de l'exécution de la commande `switch_all`

Chapitre 8

Conclusion et perspectives

En conclusion, l'hyperviseur et les composants matériels développés sont fonctionnels. L'utilisation de ceux-ci permettent le déploiement de plusieurs machines virtuelles ALMOS sur une plateforme TSAR avec HATs. Les machines virtuelles déployées sont effectivement isolées les unes des autres. L'objectif de ce stage a donc été atteint.

La suite logique de ce stage est de réaliser l'isolation complète des VMs. En effet, l'hyperviseur peut accéder à la mémoire des VMs. L'hyperviseur ne peut donc pas être dit aveugle. Cette caractéristique n'est pas atteignable aisément, elle nécessite le développement d'une procédure de démarrage où l'hyperviseur n'effectuerait aucun accès à l'intérieur de l'instance ou délèguerait ces accès à un composant dédié.

Ayant réalisé tout ce qui à été présenté dans ce rapport, il a été décidé de porter notre attention sur un problème plus prioritaire : l'utilisation d'une plateforme plus réaliste. En effet, la plateforme utilisée contient les contrôleurs de périphériques à l'intérieur de la puce. Or sur une vraie machine ces contrôleurs (TTYs, IOC, Frame Buffer, ..) se trouvent à l'extérieur de la puce. Il existe déjà une plateforme TSAR simulant les composants de cette façon : la plateforme TSAR IOB (Input/Output Bridge) 40bits. J'ai donc réalisé le portage des composants développés (MULTI_TTY_VT, MULTI_IOC), des HATs ainsi que le portage sur 40bits de l'hyperviseur sur cette plateforme.

Un autre problème n'a pas été abordé, celui de l'extinction des VMs. Une VM doit pouvoir s'éteindre d'elle même ou être éteinte par l'hyperviseur. L'extinction d'une VM doit passer par la remise à zéro de toute la mémoire des clusters qui lui ont été alloués, la désactivation des HATs, le passage en basse consommation des processeurs ainsi que la mise à jour des informations concernant les VMs et l'allocation des canaux des périphériques dans l'hyperviseur.

Les travaux réalisés durant ce stage ont conduit à la réalisation et la présentation d'un poster à l'école thématique Archi'15 ainsi qu'à une présentation au "WorkShop" CryptArchi 2015 et à la soumission d'un papier à la conférence NORCAS 2015.

Bibliographie

[1] Alain Greiner. Tsar : a scalable, shared memory, many-cores architecture with global cache coherence. In *9th International Forum on Embedded MPSoC and Multicore (MPSoC'09)*, volume 15, 2009.

[2] Ghassan Almaless. *Conception et réalisation d'un système d'exploitation pour des processeurs many-cores*. PhD thesis, Université Pierre et Marie Curie-Paris VI, 2014.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5) :164–177, 2003.

[4] Christoffer Dall and Jason 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*, pages 333–348. ACM, 2014.

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