Stage de recherche *Année scolaire 2013-2014*

Exploitation de 1 Tera-octet de mémoire physique dans ALMOS

Du 31 mars au 25 juillet 2014

François Guerret - Promotion 2011

Tuteurs École Polytechnique : **Yvan Bonnassieux,**
Henri-Jean Drouhin
Tuteur ENSTA ParisTech : **Omar Hammami**
Tuteur LIP6 : **Alain Greiner**

Laboratoire d'informatique de l'Université Paris 6 (LIP6)
Département SOC,
Équipe ALSOC,
Université Pierre et Marie Curie, Paris

Remerciements

Je tiens à remercier Alain Greiner pour m'avoir proposé ce stage et accueilli au sein de l'équipe ALSOC du LIP6, pour sa confiance et la clarté de ses explications.

Je remercie également Franck Wajsburt et Mohamed Karaoui qui m'ont guidé tout au long du stage, plus particulièrement pour m'aider à en appréhender les objectifs.

Je tiens à remercier en particulier Quentin, Cesar, Hao, Clément et Mohamed pour le temps qu'ils m'ont consacré, pour leur patience et leurs très nombreux conseils.

Merci enfin à Laetitia et Louise pour la relecture attentive de ce rapport.

Résumé

Le système d'exploitation ALMOS (*Advanced Locality Management Operating System*), actuellement en cours de développement dans le département SOC (*System on Chip*) du laboratoire d'informatique de l'Université Pierre et Marie Curie (LIP6), a été conçu pour être particulièrement efficace sur des machines many-cœurs. Un de ses principaux objectifs était de démontrer l'efficacité d'une méthode automatique de placement (et de déplacement) des données et des tâches sur les cœurs de ces machines. Il a été réalisé pour fonctionner sur des machines utilisant des processeurs 32 bits et une mémoire physique inférieure à 2 Giga-octets, les adresses physiques utilisées étaient donc de 32 bits.

L'objectif général du stage était de permettre à ALMOS d'exploiter des mémoires physiques allant jusqu'à 1 Tera-octet. Une telle quantité de mémoire vive est tout à fait envisageable à l'intérieur des machines many-cœurs, notamment telles que celles développées dans le cadre du projet TSAR (*Tera-Scale ARchitecture*) au sein duquel s'inscrit ce stage. Il nous a fallu tout d'abord modifier la phase de démarrage d'ALMOS pour que le système d'exploitation soit compatible avec un pré-loader générique, gère des adresses physiques de 40 bits et supporte une numérotation quelconque des processeurs dans l'architecture utilisée. Nous avons ensuite figé une répartition sur tous les clusters du projeté physique de l'espace virtuel du noyau. Enfin, nous avons supprimé l'image virtuelle de deux structures du noyau de tailles importantes : les tables des pages et la table des descripteurs de page.

Mots clés :

architecture many-cœurs, système d'exploitation, ALMOS, gestion de la mémoire, table des pages, table des descripteurs de page

Abstract

ALMOS (*Advanced Locality Management Operating System*) is a manycore specific operating system developed in the SOC (*System on chip*) department of the computer science loboratory of the University "Pierre et Marie Curie" (LIP6). It was made to be used on machines composed of 32 bits processors and less than 2 Giga-Bytes of physical memory, the physical addresses it handled were 32 bits addresses.

The main goal of this intership was to enable ALMOS to use up to 1 Tera-Bytes physical memories. Such an amount of random access memory are foreseen inside manycore chips such as the ones developed by the TSAR (*Tera-Scale ARchitecture*) project within which this internship has been done. First, we had to modify ALMOS boot phase in order to make it compatible with a generic pre-loader and to enable it

to deal with 40 bits physical addresses and all kind of processors numerotation in the architecture used. Then we fixed a division of the kernel space between all the clusters. Finally, to save virtual kernel space, we deleted the virtual image of two big kernel data structures : the paging tables and the table of page descriptors.

Key words :

manycore architecture, operating system, ALMOS, memory management, page table, table of page descriptors

Table des matières

Remerciements **1**

Résumé **2**

Table des figures **6**

Introduction **7**

1 Présentation de l'architecture et phase d'initialisation d'ALMOS **9**

1.1 Description de l'architecture 9

1.1.1 Description générale 10

1.1.2 Numérotation des clusters et adresses de 40 bits 11

1.1.3 Utilisation des périphériques externes, présentation du composant IOPIC 12

1.2 La phase d'initialisation d'ALMOS 14

1.2.1 Rôle du pré-loader 14

1.2.2 Le boot-loader et la phase d'initialisation avec un seul processeur actif 14

1.2.3 Fin de l'initialisation d'ALMOS 15

2 Problèmes liés à la représentation virtuelle de l'espace mémoire physique **18**

2.1 Présentation 18

2.1.1 La table des pages 19

2.1.2 La table des descripteurs de page physique 20

2.1.3 Les gestionnaires de la mémoire 21

2.2 Une représentation de tout l'espace physique dans l'espace virtuel noyau 22

2.2.1 Un problème de passage à l'échelle 22

2.2.2 Projeté de l'espace virtuel du noyau dans tous les clusters 23

2.3 Taille des structures du noyau 23
2.3.1 À l'échelle de la puce 23
2.3.2 A l'échelle d'un cluster 24
2.3.3 Solutions envisagées 24
2.4 Démarche suivie 25

3 Implémentation des solutions retenues 27
3.1 Considérations générales 27
3.2 Séparation des différents domaines de la mémoire physique 28
3.3 Les tables des pages 29
3.4 La table des descripteurs de page 29

Conclusion 31

Bibliographie 32

A Exemple de fonction utilisant l'adressage physique direct 33

Table des figures

1.1 Représentation d'une puce de type `tsar_generic_leti` à 10 clusters. 10
1.2 Représentation de la correspondance entre espace virtuel et espace physique à la fin de la phase d'initialisation. 17

2.1 Traduction d'une adresse virtuelle en adresse physique grâce à la table des pages, d'après [2] 19
2.2 Représentation de la correspondance entre espace physique et espace virtuel 26

Introduction

Le projet TSAR (*Tera-Scale ARchitecture* [2]) auquel participe l'équipe d'Architecture et Logiciels pour Systèmes Embarqués sur Puce (ALSOC) du département SOC (*System on chip*) du laboratoire d'informatique de l'Université Pierre et Marie Curie (LIP6) a pour objectif la définition d'une architecture passant à l'échelle à mémoire partagée cohérente et le développement des prototypes virtuel correspondant. Le système d'exploitation ALMOS (*Advanced Locality Management Operating System* [1]), développé principalement par Ghassan Almaless et Franck Wajsburt [5, 4], a été conçu pour gérer de manière automatique le placement et le déplacement des processus et des données sur les cœurs d'une machine multi-cœurs ou many-cœurs. Cette gestion a pour objectif de maximiser la localité des accès mémoire, notamment des accès mémoire d'une tâche parallélisée exécutée par plusieurs threads au sein d'un même processus.

Dans le projet TSAR, le choix de l'utilisation de processeurs 32 bits a été fait pour des raisons de consommation d'énergie. Une de ses conséquences est que l'espace virtuel des processus exécutés sur ces puces est limité à 4 Go. Cette particularité empêchait le système d'exploitation ALMOS d'utiliser la totalité de la mémoire physique à sa disposition, celle-ci étant potentiellement très importante dans les architectures many-cœurs. Par exemple une puce de 256 clusters contenant chacun 4 Go de mémoire possède une mémoire vive de 1 To alors qu'ALMOS ne pouvait n'en exploiter que 2 Go. Nous développerons cet exemple dans la suite de ce document afin de fournir des évaluations chiffrées des phénomènes constatés.

L'objectif général de notre étude est de permettre l'exploitation de la grande mémoire vive des puces many-cœurs - des puces TSAR notamment - par ALMOS. Il nous a fallu tout d'abord modifier la phase de démarrage d'ALMOS pour que le système d'exploitation soit compatible avec un pré-loader générique, gère des adresses physiques de 40 bits et supporte une numérotation quelconque des processeurs dans l'architecture utilisée. Nous avons ensuite figé une répartition sur tous les clusters du projeté physique de l'espace virtuel du noyau. Enfin, nous avons supprimé l'image virtuelle de deux structures du noyau de tailles importante : les tables des pages et la table des descripteurs de page.

Dans un premier temps, afin de situer le contexte de notre étude, nous décrirons succinctement une architecture many-cœur en prenant pour exemple la puce `tsar_generic_leti` qui a constitué la machine cible principale sur laquelle nous devions faire fonctionner le système d'exploitation ALMOS. Nous décrirons à cette occasion les grandes lignes de la phase d'initialisation d'ALMOS, à laquelle nous avons dû apporter un certain nombre de modifications pour permettre ce portage. Nous détaillerons ensuite les différents problèmes posés par l'implémentation initiale d'ALMOS vis à vis de la gestion des espaces mémoire physique et virtuel quand la taille de la mémoire vive à gérer augmente. Enfin, nous détaillerons les solutions que nous avons ou sommes en train d'implémenter pour répondre à ces problèmes.

Veuillez noter que le présent rapport n'est pas la version définitive, une dernière partie présentant une évaluation de l'impact de nos modifications sur les performances du système d'exploitation sera réalisée à la fin du stage, c'est à dire pour la dernière semaine de juillet 2014.

Partie 1

Présentation de l'architecture et phase d'initialisation d'ALMOS

Afin de présenter le cadre de notre étude, nous allons dans un premier temps décrire ce qu'est une architecture many-cœurs en présentant en particulier l'architecture de la puce `tsar_generic_leti`. Un des objectifs de nos travaux étant de permettre au système d'exploitation ALMOS de fonctionner sur cette puce, nous préciserons pour chacune de ses spécificités les modifications que nous avons dû apporter au noyau d'ALMOS. Dans un deuxième temps, nous détaillerons les grands principes de la phase d'initialisation - communément appelées phase de *boot* - du système d'exploitation et les modifications principales que nous lui avons apportées. En effet, la phase d'initialisation est liée par de nombreux aspects à l'architecture sur laquelle le système d'exploitation est exécuté.

1.1 Description de l'architecture

Nous avons testé nos travaux grâce à un prototype virtuel TSAR précis au cycle et au bit prêt. Ce prototype décrit de manière fidèle le comportement de la puce `tsar_generic_leti` sur laquelle devait pouvoir être utilisé ALMOS. Pour faciliter la compréhension des paragraphes suivants, on pourra se référer à la Figure 1.1 qui représente un exemple de ce type de puce.

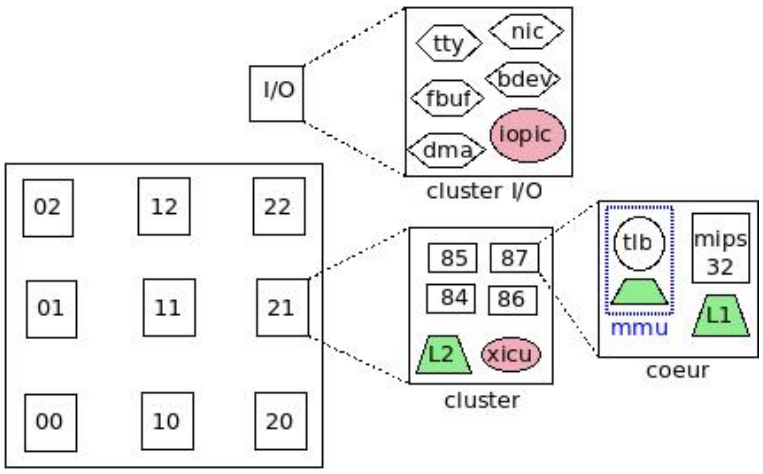

FIGURE 1.1 – Représentation d'une puce de type `tsar_generic_leti` à 10 clusters.

1.1.1 Description générale

Cœurs et clusters

La puce many-cœur `tsar_generic_leti` se compose d'un grand nombre de cœurs regroupés en clusters. Un cœur contient un processeur, un cache L1 et une MMU (*Memory Management Unit*), elle-même composée d'une TLB (*Translation Look-aside Buffer*) et d'un cache. Chaque cluster contient un certain nombre de cœurs - quatre pour ce qui est de la puce en cours de fabrication sur laquelle doit être utilisé le système d'exploitation. Les clusters peuvent contenir également des périphériques.

Dans la puce `tsar_generic_leti`, tous les clusters sont dotés d'un gestionnaire d'interruption (XICU) et d'un cache L2 (et de son contrôleur de cache). La cohérence des caches est assurée au niveau L2. Le cluster (0,0) [1] contient en plus un terminal (TTY) et un contrôleur de disque (BDEV). Enfin, le cluster dit cluster I/O (entrée/sortie) ne contient pas de processeurs mais contient un certains nombre de périphériques externes : un terminal, un contrôleur de disque, un tampon de trame (*framebuffer*, FBUF), un contrôleur DMA (*Direct Memory Access*), un contrôleur d'interface réseau (NIC) et un composant de traduction d'interruptions, l'IOPIC. L'utilisation des périphériques externes et le fonctionnement de l'IOPIC sont détaillés dans la sous-section 1.1.3.

Notons enfin que les clusters sont reliés entre eux par trois réseaux indépendants : le réseau direct, le réseau de cohérence et le réseau RAM.

1. Se référer à la sous-section 1.1.2 pour le sens de cette numérotation

Le registre d'extension d'adresse

Les processeurs MIPS32 utilisés dans l'architecture TSAR sont dotés d'un registre d'extension d'adresse (le registre $24 du coprocesseur 2) qui permet, lorsque la MMU du processeur est désactivée, d'ajouter de manière systématique 8 bits d'extension d'adresse aux adresses 32 bits manipulées par le processeur. Grâce à ce registre, nous disposons d'un moyen simple d'écrire ou de lire à n'importe quelle adresse physique de la mémoire vive. L'architecture TSAR permet donc d'adresser directement, c'est à dire sans devoir passer par une représentation virtuelle temporaire, jusqu'à 1 To de mémoire physique (adresses physiques de 40 bits).

Modélisation du cache L3

Le cache L3 dont est doté l'architecture TSAR est modélisé par une certaine quantité de mémoire vive (RAM) logiquement partagée mais physiquement distribuée. La mémoire vive est répartie de manière *a priori* équitable entre tous les clusters. Nous avons supposé que c'était effectivement le cas notamment pour implémenter la solution au problème détaillé en 2.2.

1.1.2 Numérotation des clusters et adresses de 40 bits

Numérotation

Chaque cluster est désigné par un couple de coordonnées (x,y) désignant la position du cluster dans le mesh. L'identifiant global d'un processeur, inscrit en dur dans la puce, est défini par

$$gid = (identifiant\_cluster * cpu\_nr) + lid$$

où `cpu_nr` désigne le nombre de processeurs (c'est à dire le nombre de cœurs) par cluster et `lid` l'identifiant local du processeur dans le cluster auquel il appartient. La Figure 1.1 illustre cette numérotation.

Les huit bits de poids forts d'une adresse physique sont utilisés pour désigner le cluster qui contient la portion de mémoire où l'adresse se trouve. La coordonnée x du cluster constitue les 4 bits de poids forts et la coordonnée y les 4 bits de poids faibles. La puce contient donc au maximum 256 clusters. L'espace mémoire physique peut être discontinu, soit si le nombre de cluster est inférieur à 256 (puisque la numérotation des cluster est discontinue), soit si la quantité de mémoire physique qu'ils contiennent est inférieure à 4 Giga-octets. Le cas où la mémoire physique est continue correspond donc au seul cas où la quantité de mémoire physique est maximale et égale à 1 Tera-octet (cas d'une puce à 256 clusters et 4

Go de mémoire par cluster). Notons que pour des raisons techniques, la puce `tsar_generic_leti` ne peut contenir que 128 cluster en raison de la présence d'un (ou éventuellement plusieurs) cluster(s) I/O.

Modifications apportées au système d'exploitation

Le système d'exploitation ALMOS faisait l'hypothèse que les indices de numérotation des clusters et des processeurs étaient continus. Nous avons donc rajouté un niveau d'indirection en isolant la numérotation employée par le système d'exploitation de celle spécifique à l'architecture.

Les deux tableaux de traduction des identifiants des clusters construits à cette fin (traduction de l'identifiant architecture en identifiant continu et inversement) sont initialisés lors de la première partie de la phase d'initialisation où un seul processeur est actif. Ils sont utilisés lors du réveil des autres processeurs (se référer à 1.2) et utilisés pour toutes les traductions à chaque fois qu'un processus a besoin de connaître l'identifiant du processeur (donc du cluster, la formule indiquée au début de cette sous-section étant valable quelque soit la numérotation de cluster choisie) sur lequel il est en train de s'exécuter. Nous avons réalisé une copie locale de ces tableaux sur tous les clusters afin de ne pas créer un goulot d'étranglement sur le cluster de boot [2].

1.1.3 Utilisation des périphériques externes, présentation du composant IOPIC

Les clusters sans processeurs, qu'ils soient vides ou qu'ils contiennent des périphériques, n'étaient pas gérés par ALMOS. Le niveau d'indirection rajouté pour la numérotation nous a permis de gérer les clusters vides (cela revient à gérer un indice de numérotation discontinu pour les clusters). Pour utiliser les périphériques des clusters sans processeurs (clusters I/O), nous avons désigné un cluster en particulier dont un des processeurs sera en charge de les initialiser. Ce cluster est choisi par l'utilisateur (ou le pré-loader), qui a tout intérêt à choisir le cluster le plus proche du ou des clusters I/O afin de minimiser l'encombrement des communications sur le réseau.

La communication entre les périphériques des clusters I/O et les processeurs est rendue possible grâce à un composant particulier dont nous avons dû créer le pilote : l'IOPIC. Pour en détailler le fonctionnement, il nous faut d'abord décrire celui des gestionnaires d'interruptions dont nous avons également modifié le pilote.

2. On désigne par « cluster de boot » le cluster au sein duquel se trouve le « processeur de boot », c'est à dire le premier processeur à exécuter le `boot-loader` lors de l'initialisation de la machine.

Le gestionnaire d'interruption

Il existe trois types d'interruptions : les interruptions matérielles (HWI pour *Hardware Interrupt*), les interruptions levées par les timer (PTI pour *Programmable Timer Interrupts*) et les interruptions logicielles (WTI pour *Write-Triggered Interrupt*). Le composant XICU est chargé, dans chaque cluster, de centraliser ces différents types d'interruptions et de lever une interruption (IRQ) vers chaque processeur du cluster concerné par au moins une des interruptions reçues. Dans le cas des WTI, c'est l'écriture dans un registre particulier de l'XICU qui lève une interruption. Notons que la valeur écrite peut être lue et utilisée par le processeur interrompu.

L'IOPIC

Dans les clusters dotés de processeurs, les périphériques lèvent des interruptions matérielles (HWI) vers le composant XICU propre au cluster. Mais comme il n'existe pas de réseau pour les interruptions dans la puce, les périphériques du cluster I/O (qui, rappelons-le, ne contient pas de processeur) ne peuvent pas lever d'éventuelles HWI vers l'XICU d'un cluster avec processeurs. Le rôle de l'IOPIC est de traduire les interruptions HWI levées par les périphériques du cluster I/O en WTI qui vont être envoyées vers les XICU des différents clusters. Les WTI utilisent le réseau direct de la puce.

Détaillons, à titre d'exemple, le rôle de la fonction d'initialisation de l'IOPIC que nous avons implémentée. Pour chaque périphérique d'un cluster sans processeurs, cette fonction effectue plusieurs opérations :

- elle choisi le processeur qui aura à gérer les interruptions du périphérique [3]
- elle indique à l'IOPIC à quelle adresse il doit envoyer une WTI quand il reçoit la HWI du périphérique en question (cette adresse est celle d'un des registres de l'XICU du cluster contenant le processeur précédemment choisi),
- elle indique au pilote de l'XICU quel est le gestionnaire d'interruption (*handler*) à appeler quand cette WTI est reçue,
- elle indique à l'XICU quelle est le processeur à interrompre quand cette WTI est reçue.
- elle crée l'inode du périphérique dans le système de fichier.

Le portage d'ALMOS sur la puce que nous avons décrite a nécessité un certains nombre de modifications, notamment dans la phase d'initialisation du système d'exploitation. Nous décrivons cette phase d'initialisation telle que nous l'avons modifiée dans la section suivante.

3. Ce choix est spécifique à l'architecture, dans le cas de `tsar_generic_leti`, on a choisi de répartir les WTI à raison de deux WTI par processeur en favorisant les clusters proches du cluster I/O.

1.2 La phase d'initialisation d'ALMOS

Un des objectifs du stage a consisté à rendre la phase d'initialisation compatible avec un pré-loader générique déjà utilisé pour d'autres systèmes d'exploitations sur les machines TSAR. Le pré-loader est en effet présent « en dur » dans la ROM des machines et l'on veut pouvoir utiliser les puces avec différents systèmes d'exploitations.

1.2.1 Rôle du pré-loader

Le pré-loader est exécuté par tous les processeurs lors de la mise sous tension de la machine. Tous les processeurs initialisent alors leur pile, démasquent les interruptions qui leur permettront d'être réveillés par le processeur de boot et passent en régime basse consommation (tous sauf un). Un des processeur - le processeur de boot - charge alors le code et les données du boot-loader, le code et les données du noyau et une description de l'architecture depuis le disque où ils sont stockés jusque dans la RAM du cluster auquel ce processeur appartient (le cluster de boot). Le processeur de boot exécute ensuite le code du boot-loader.

Puisque les processeurs manipulent des adresses de 32 bis, ce scénario n'est possible que si le cluster de boot est le cluster (0,0). Dans l'objectif de gérer une tolérance au pannes (hypothèse d'un cluster défectueux), il faut rendre possible l'initialisation à partir d'autres clusters. Pour cela, il est nécessaire que tous les processeurs placent, lors de l'exécution du pré-loader, l'identifiant du cluster de boot dans leur registre d'extension d'adresse. Ainsi, tous pourront lire les informations qui ont été placées dans la RAM de ce cluster à leur réveil (rappelons que leur MMU est toujours désactivée à cet instant).

1.2.2 Le boot-loader et la phase d'initialisation avec un seul processeur actif

Contrairement au rôle qu'a le boot-loader dans Linux (on pourra se référer à [7], par exemple), celui d'ALMOS ne charge pas une image du noyau dans la RAM, cela étant déjà réalisé par le pré-loader. Cependant, une de ses première tâches est de déplacer le code et les données du noyau afin de les aligner sur une grande page physique (espace physique continu de 2 Mo). En effet, le premier *mapping* réalisé par le processeur de boot va représenter le noyau dans des grandes pages virtuelles.

Le rôle du processeur de boot est de permettre l'initialisation de tous les autres processeurs. Il construit à cette fin une table (« table de boot ») où chacun des processeurs pourra trouver l'adresse virtuelle du sommet de sa pile, la valeur du *ptpr* (c'est à dire l'adresse de base de la table des pages qu'a construite le processeur de boot à un décalage près) et l'adresse de base de son espace virtuel noyau, entre autres. Cette première étape a lieu en deux temps : tout d'abord, le boot-loader prépare l'activation de la MMU, en

réalisant notamment le premier *mapping*, et le passage dans le noyau. Puis le processeur de boot active sa MMU et passe dans le code du noyau où il réalise un certains nombre d'initialisations générales (verrous, tableaux de traductions des numérotations, construction de la table de boot, etc.). Enfin, il retourne dans le code du boot-loader pour réveiller, à l'aide de l'envoi de WTI tous les autres processeurs. La valeur inscrite dans le registre des XICU est l'adresse d'entrée dans le code du boot-loader. Dès la WTI reçue, les processeurs exécutent le code à l'adresse reçue (ce procédé permet une certaine souplesse puisque le processeur de boot est ainsi libre de déplacer le code du boot-loader si le pré-loader ne l'a pas chargé à l'endroit prévu). Le fait de retourner dans le code du boot-loader devra probablement être changé mais ce choix s'explique par le fait que le réveil des processeurs est spécifique à l'architecture utilisée et à la manière dont les processeurs ont été mis en attente.

La correspondance entre l'espace virtuel et l'espace physique du cluster de boot à la fin de la phase d'initialisation avec un seul processeur actif est proche de celle représentée à la Figure 1.2. Cette figure présente un état des lieux à la fin de l'initialisation.

1.2.3 Fin de l'initialisation d'ALMOS

Une fois tous les processeurs autres que le processeur de boot réveillés, ils activent immédiatement leur MMU et passent rapidement dans le noyau. Pour ce faire, ils ont notamment besoin d'accéder à l'entrée de la « table de boot » qui les concerne. Ils doivent donc déterminer l'indice du cluster auquel ils appartiennent dans la numérotation continue utilisée par ALMOS. C'est pour cela que les tableaux de traduction des identifiants des clusters doivent être construits par le processeur de boot.

La fin de l'initialisation se fait en deux étapes : un processeur par cluster initialise chaque cluster puis tous les processeurs se placent en état d'attente d'évènements sauf un qui est chargé de construire le système de fichier virtuel et de lancer l'exécution d'un fichier d'initialisation.

Initialisation des clusters

Un processeur par cluster initialise :

- la structure décrivant ce cluster (`cluster_s`),
- le gestionnaire de l'espace physique du cluster (il construit donc la table des descripteurs de page de l'espace physique du cluster)
- la table des structures décrivant les cœurs du cluster,
- le *Kernel Cache Manager*,
- le gestionnaire du tas,

- tous les périphériques du cluster (initialisation du pilote, attribution d'une page virtuelle, initialisation des correspondances entre HWI (reçues des périphériques du cluster) et IRQ (transmises vers les processeurs) dans l'XICU et création de l'inode associé au périphérique).

Passage en fonctionnement courant

Une fois ces étapes d'initialisation terminées, tous les processeurs initialisent et exécutent (c'est à dire placent dans leur *scheduler*) un thread d'attente (`thread_idle`) et un thread chargé de la gestion des évènements (`thread_event_manager`). Enfin, un des processeurs initialise le système de fichier virtuel puis charge depuis le disque et exécute le code de l'exécutable binaire `init`. Le système d'exploitation est alors initialisé et prêt à fonctionner.

Deux des points essentiels de la phase d'initialisation sont le placements des données et le premier *mapping*. La Figure 1.2 représente l'espace virtuel du processus exécuté par le processeur de boot et l'espace physique du cluster de boot à la fin de la phase d'initialisation.

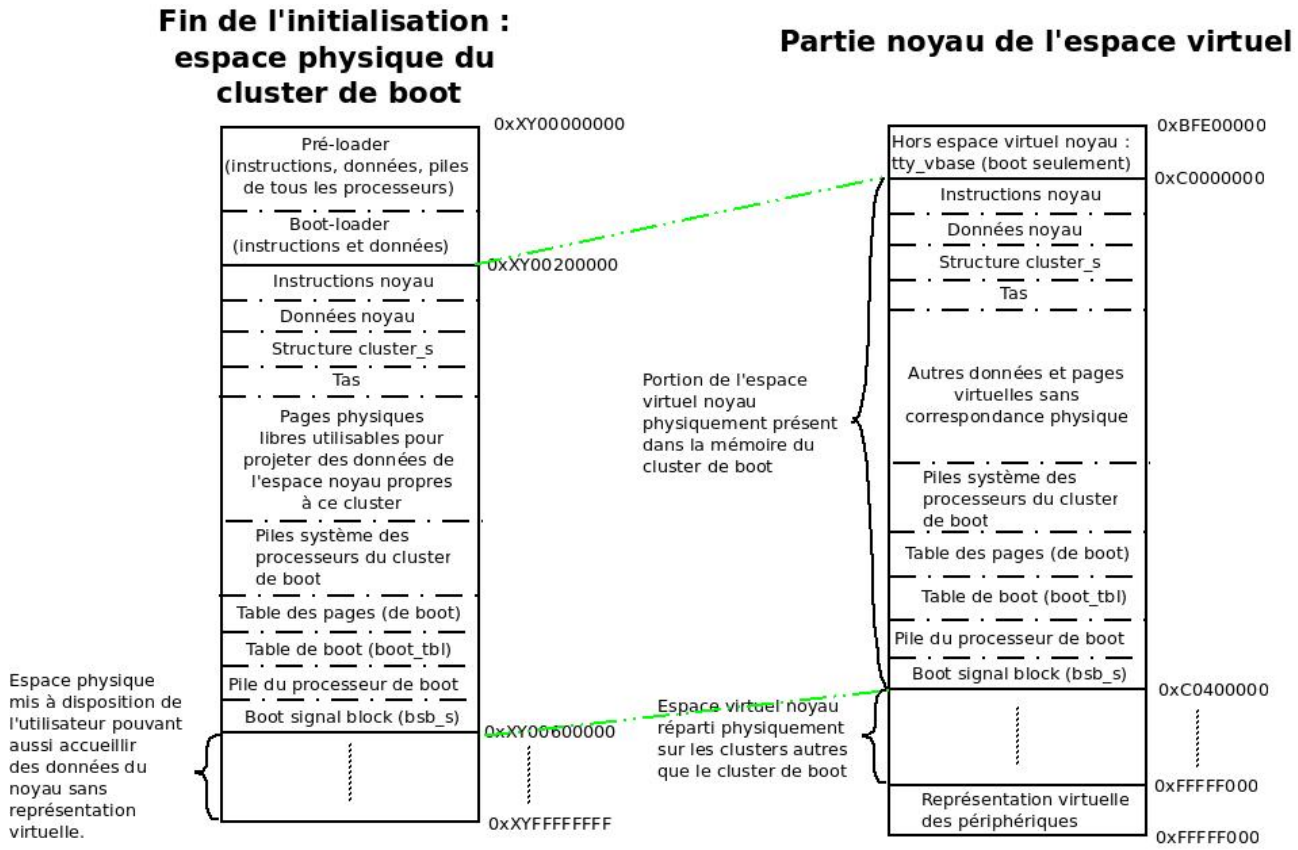

FIGURE 1.2 – Représentation de la correspondance entre espace virtuel et espace physique à la fin de la phase d'initialisation. On s'est placé dans le cas d'une portion de l'espace virtuel noyau « dédiée à un cluster » de 4 Mo (on détaillera la signification de ces termes en 2.2).

Partie 2

Problèmes liés à la représentation virtuelle de l'espace mémoire physique

ALMOS présentait essentiellement deux problèmes liés à la représentation virtuelle de l'espace mémoire physique. En premier lieu, le système d'exploitation représentait tout l'espace physique dans l'espace virtuel noyau. Ce dernier ayant été limité à 2 Go, cela limitait le fonctionnement d'ALMOS à des machines ayant une mémoire vive inférieure à 2 Go. Le second tenait à la taille trop importante de certaines structures du noyau comme les tables des pages et la table des descripteurs de page physique. Nous allons tout d'abord présenter ces deux structures afin de préciser la relation entre espace physique et espace virtuel, puis nous détaillerons les deux problèmes que nous venons de soulever. Enfin, nous donnerons un aperçu théorique des solutions apportées. La partie 3 décrit quant à elle l'implémentation que nous avons réalisé de ces solutions.

2.1 Présentation

Les deux structures qui nous intéressent dans cette section sont la table des pages et la table des descripteurs de page physique. Nous les présentons brièvement dans les paragraphes suivants telles qu'elles avaient été initialement implémentées dans ALMOS.

On désigne par « page physique » un espace de la mémoire physique dont les adresses sont consécutives, on utilisera les termes « petites pages » et « grandes pages » pour désigner les deux types de pages virtuelles gérées par ALMOS. Une page physique selon cette définition est décrite par la structure `page_s`, ces espaces sont décomposés en petites ou grandes pages lors de leur allocation.

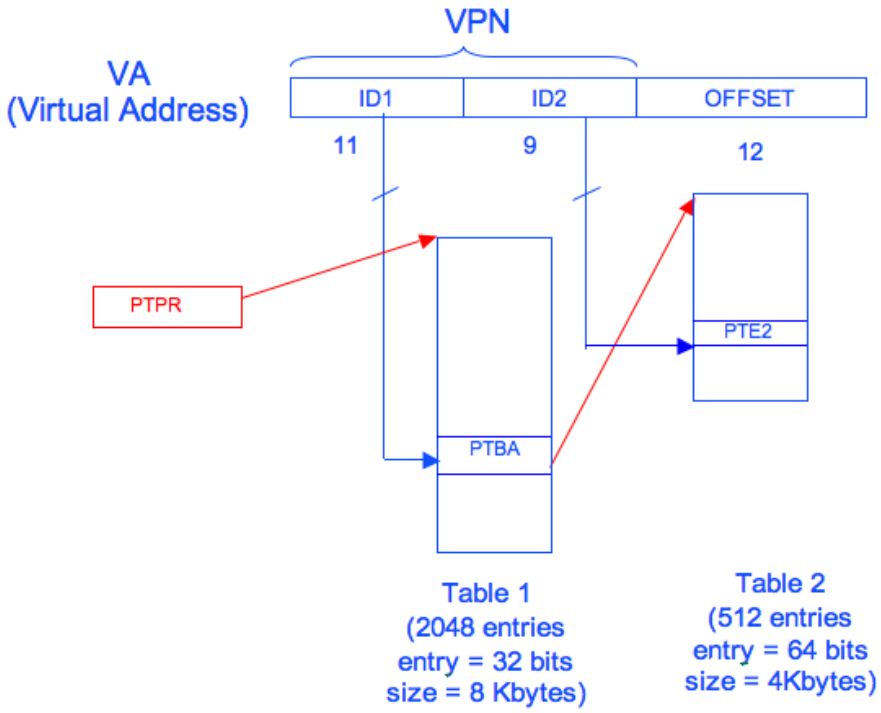

FIGURE 2.1 – Traduction d'une adresse virtuelle en adresse physique grâce à la table des pages, d'après [2]

2.1.1 La table des pages

La table des pages est une table à deux niveaux, elle était contenue et manipulée dans l'espace virtuel du noyau. Les entrées de la table de premier niveau contiennent les adresses physiques (le *ppn1*[1]) de portions de l'espace physique de la taille d'une « grande page » (2 Mo) ou des pointeurs vers des tables de second niveau. Les entrées de la table de deuxième niveau contiennent, quant à elles, les adresses physiques (le *ppn2*) de portions de l'espace physique de la taille d'une « petite page » (4 Ko). Une table des pages est propre à un espace virtuel donné, chaque application a donc sa propre table des pages. La figure 2.1 rappelle le procédé par lequel on peut obtenir une adresse physique à partir de l'adresse virtuelle qui la désigne. Dans ALMOS, la table de premier niveau est contenue dans le tableau `pgdir`. L'adresse virtuelle de ce tableau était stockée dans un champ du gestionnaire de mémoire physique (`pmm`) propre à chaque cluster. L'adresse physique de la première case de ce tableau correspond donc à la valeur (à un décalage de 13 bits près) fournie comme *ptpr* au processeur. Les tables de second niveau sont dispersées dans l'espace

1. Le ppn d'une adresse physique est obtenu en réalisant un décalage de 12 bits (de la puissance de deux correspondant à la taille d'une petite page) vers la droite. Le ppn permet donc d'identifier à quelle petite page une adresse physique appartient.

virtuel (et physique) du cluster dont elles désignent les pages physiques (les pages physiques les contenant sont attribuées à la volée, selon les besoins d'une application).

2.1.2 La table des descripteurs de page physique

Les descripteurs de page physique (de type `page_s`) contiennent des informations sur le statut libre ou utilisé d'une page, sur le nombre d'applications utilisant cette page et sur sa taille notamment. La taille d'une page est caractérisée par son ordre, `order`, telle que $taille = 4Ko * 2^{order}$, les tailles s'échelonnent de 4 Ko (ordre 0, taille d'une petite page (`PAGE`)) à 2 Mo (ordre 9, taille d'une grande page (`HUGE_PAGE`)).

La table des descripteurs est toujours d'une taille maximale et ne peut être réduite. Le cas où le nombre de descripteurs de page physique libre est maximal correspond au cas où la fragmentation externe de la mémoire est maximale, soit quand une « petite page physique » sur deux est libre. On pourrait donc espérer, avec une structure de liste chaînée, que les descripteurs de page physique prennent deux fois moins d'espace physique que l'espace qu'ils occupent actuellement. Cependant, un choix a été fait pour que les descripteurs de page physique décrivent l'espace physique quelqu'en soit l'état (alloué ou non).

Organisation des descripteurs en tableau

Les descripteurs de page physique sont d'un nombre fixe parce qu'ils décrivent l'espace physique dans son ensemble, que les pages soient occupées ou non. Cette indépendance implique que chaque descripteur doit décrire une portion de l'espace physique correspondant à la taille minimale pouvant être allouée, soit une petite page de 4 Ko. Le nombre fixe de descripteurs autorise une organisation en tableau, qui permet notamment :

- de gérer selon une implémentation classique de l'algorithme *buddy* la table des descripteurs (dans le code, se référer aux fonctions `ppm_alloc_page` et `ppm_free_pages_nolock` qui implémentent les fonctionnalités au niveau de la mémoire physique de *malloc* et *free*),
- d'accéder en temps constant à l'état de n'importe quelle portion d'espace physique (une liste chaînée nécessite un parcours),
- de simplifier la désignation de l'espace physique décrit par un descripteur, en exploitant notamment le fait que la différence entre l'adresse virtuelle du descripteur et l'adresse virtuelle de base de la table des descripteurs correspond à un index désignant une petite page dans l'espace physique.

Si cette implémentation a un coût en terme d'espace mémoire occupé (par rapport à une seule liste des pages libres), celui-ci est compensé par un gain en temps (à chaque libération d'un espace mémoire) qui

croît avec la taille de la mémoire à gérer et par la plus facile implémentation de l'algorithme *buddy*[2]. Nous n'avons donc pas essayé de supprimer cette organisation en tableau de la table des descripteurs, ce que nous aurions pu tenter pour économiser de l'espace mémoire.

Organisation des descripteurs en liste

Un des champs des descripteurs de page physique `page_s` est un pointeur vers le descripteur de la pages libre suivante de la même taille que la page qu'ils décrivent. Ils contiennent également un pointeur vers le descripteur de la page précédente de même taille. Chaque descripteur appartient ainsi à une liste doublement chaînée de descripteurs de page faisant une taille donnée (ayant un ordre donné). Le gestionnaire de pages physiques contient un tableau de pointeurs qui désignent chacun le premier descripteur de page d'un ordre donné (le tableau a donc dix entrées).

Cette structure de liste est en fait utilisée comme une pile (*Last In First Out*) : lorsqu'une page est libérée elle prend la place de la racine dans la liste des pages de la taille de l'espace libre contiguë obtenu ; et la racine d'une liste est le descripteur de la prochaine page d'une taille donnée à allouer (entièrement ou en partie). Il y a au moins deux avantages à cette organisation : cela favorise la localité des accès mémoire (dans le cache des données qui doit charger les descripteurs) et surtout, cela fourni un accès en temps constant à un descripteur de page libre (il faudrait sinon parcourir la table).

Ainsi, lors de l'allocation d'une portion d'espace physique, la structure de liste permet un accès immédiat à une page libre et, lors de la libération d'une portion de l'espace physique, la structure de tableau permet un accès immédiat au descripteur de page concerné.

2.1.3 Les gestionnaires de la mémoire

Dans ce paragraphe, nous présentons de manière très concise les gestionnaires de la mémoire dans ALMOS. Chaque application (« tâche ») est dotée d'un gestionnaire de mémoire virtuelle (`vmm`) qui possède lui-même un gestionnaire de mémoire physique `pmm` chargé de la correspondance entre adresses virtuelles et adresses physique (il contient entre autre l'adresse du tableau `pgdir` de l'application en cours d'exécution sur le processeur). Pour attribuer ou libérer des pages physiques, les fonctions du système d'exploitation utilisent le gestionnaire de mémoire du noyau (`kmem`) qui fait lui-même appel au gestionnaire de pages physiques (`ppm`). Ce dernier contient effectivement le tableau des pointeurs vers les listes de descripteurs de page physique libre (champ `free_pages`) et l'adresse de base de la table des descripteurs de page

2. La structure de tableau permet des accès immédiat aux différentes entrées de la table à mettre à jour à chaque allocation ou libération de mémoire.

physique (`pages_tbl`). Le `ppm` est propre à chaque cluster et est entièrement maître de la gestion de la mémoire physique du cluster auquel il appartient. Toute tâche voulant demander l'attribution d'une page dans un cluster donné doit réaliser une prise de verrou sur le gestionnaire de pages physiques (`ppm`) qui en a la charge.

2.2 Une représentation de tout l'espace physique dans l'espace virtuel noyau

2.2.1 Un problème de passage à l'échelle

Dans ALMOS, toute adresse physique était représentée par une adresse virtuelle dans l'espace virtuel noyau. Cette caractéristique était exploitée pour ne pas avoir à gérer directement des adresses physiques. En effet, des pages physiques à l'usage de l'utilisateur pouvaient ainsi être allouées et manipulées sans que des adresses virtuelles de l'espace utilisateur ne leur soient consacrées. Pour permettre au système d'exploitation de gérer un espace physique supérieur à 2 Go, il nous a fallu supprimer cette correspondance entre adresses physiques et adresses virtuelles du noyau.

Les deux cas à gérer étaient les suivants : soit le système d'exploitation allouait des pages pour l'utilisateur de manière temporaire, soit il les allouait de manière permanente et stockait les adresses virtuelles noyaux correspondantes pour une utilisation ultérieure. Pour résoudre ce problème, nous avons séparé la gestion de « l'espace physique noyau » (on désigne par ce terme les pages physiques représentées (*mappées*) dans l'espace virtuel du noyau) de celle de l'espace physique à disposition de l'utilisateur. Notons que l'utilisateur peut toujours allouer des pages dans l'espace physique noyau si l'espace physique utilisateur est entièrement occupé. Dans le premier cas, il a suffit, à chaque fois, d'allouer une page physique dans l'espace noyau puisqu'elle est libérée rapidement. Dans le second cas, nous avons soit procédé de même (mais cette solution a ses limites puisque la taille de l'espace virtuel noyau est problématique comme on l'expliquera dans la section suivante), soit réalisé la gestion directement à l'aide d'un adressage physique. Cette gestion en adressage physique a été implémentée à l'aide de fonctions similaires à celles que nous décrirons dans la partie 3.

D'autre part, il n'y avait pas de seuil fixé limitant la portion de l'espace virtuel accordé au noyau projetée physiquement dans un cluster donné. Notre premier objectif a donc été de fixer une répartition du projeté physique de l'espace virtuel du noyau dans tous les clusters.

2.2.2 Projeté de l'espace virtuel du noyau dans tous les clusters

Nos premières modifications ont consisté à attribuer à chaque cluster une certaine part de l'espace virtuel du noyau dans lequel il puisse stocker notamment les structures qui lui sont propres.

Chaque application possède un espace virtuel utilisateur qui lui est propre, mais l'espace virtuel du noyau est commun à toutes les applications (s'exécutant sur tous les clusters). Nous ne détaillerons pas le statut particulier du code des réplica noyau (se référer à [3] pour leur rôle dans le système d'exploitation) car celui-ci est de petite taille, son impact est donc négligeable. Le projeté de l'espace virtuel noyau [3] en mémoire physique a été réalisé de manière répartie sur les différents clusters afin de favoriser les accès locaux en mémoire et d'éviter la sollicitation d'un seul cluster (goulot d'étranglement) [4]. Ainsi l'espace virtuel disponible sur un cluster est de $\frac{\text{taille de l'espace virtuel noyau}}{\text{nombre de clusters}}$ soit 4 Mo dans le cas d'une puce à 256 clusters et d'un espace virtuel noyau de 1 Go.

Dans une première étape, nous avons ainsi permis à ALMOS de pouvoir s'exécuter dans des architectures présentant une quantité importante de mémoire vive. Ce n'est qu'ensuite que nous avons fait en sorte que le système d'exploitation puisse effectivement exploiter les ressources de mémoire physique à sa disposition.

2.3 Taille des structures du noyau

Il est possible de fournir deux points de vue sur le problème de la taille des structures de données du noyau : à l'échelle de la puce et à l'échelle d'un cluster. Le premier permet de comprendre la source du problème et le second permet d'en appréhender les implications et facilite la compréhension de la solution proposée dans la partie suivante.

2.3.1 À l'échelle de la puce

La taille maximale de l'espace virtuel du noyau était de 2 Go dans ALMOS. En même temps que nous avons réalisé la division de cet espace expliquée en 2.2.2, nous l'avons réduit à 1 Go (pour augmenter la taille de l'espace virtuel utilisateur). Comme les tables des pages étaient gérées dans l'espace virtuel noyau, leur taille trop élevée posait problème. En effet, les entrées de la table de premier niveau sont composées d'un mot de 32 bits et celles de deuxième niveau de deux mots de 32 bits, la table a donc au plus une

3. Pour plus de concision dans la rédaction, « l'espace virtuel noyau » désigne « la portion de l'espace virtuel accordée au noyau ».

4. En particulier, on comprend aisément qu'il est nécessaire que les tables des pages et celle des descripteurs de page soient contenues dans l'espace physique du cluster qu'elles concernent

taille de $(2^{32-21} + 2 * 2^{32-12}) * 4 \simeq 8Mo$ par processus. A raison d'une application par cluster - ce qui correspond à un cas relativement favorable - ces tables prenaient jusqu'à 2 Go de l'espace virtuel, elles nécessitaient donc de réserver deux fois la taille de l'espace virtuel consacré au noyau !

Notons que la taille d'une table a été calculée dans le pire des cas, qui n'est jamais atteint, ne serait-ce que parce que le code du noyau et ses données sont projetés dans des grandes pages, par exemple. Cependant, même si une table ne fait qu'un quart de cette taille maximale, il n'est pas invraisemblable d'imaginer qu'un cluster puisse contenir quatre processus en exécution ou en attente (notamment s'il s'agit d'un cluster à quatre cœurs), le problème est alors inchangé.

La table des descripteurs de page physique posait encore davantage de problèmes car sa taille atteint $(14*2^{28})*4 \simeq 14\,Go$. En effet, un descripteur de page physique a une taille de 14 mots de 32 bits et la table a une taille fixe. Il s'agissait en fait également d'un problème de passage à l'échelle puisque le problème ne se posait que pour un nombre suffisamment élevé de clusters (ou pour des mémoires très importantes dans chaque cluster).

2.3.2 A l'échelle d'un cluster

Chaque cluster est responsable de la gestion de la mémoire physique qu'il contient (via son gestionnaire de page physique (`ppm`)). Comme expliqué en 2.2.2, la portion de l'espace virtuel « dédiée » à un cluster est très réduite, et on voulait y placer la table des pages et la table des descripteurs des pages physiques de ce cluster. La traduction locale du problème global énoncé dans le paragraphe précédent est que cela était impossible : dans notre exemple, la taille de l'espace virtuel noyau « dédié à un cluster » vaut 4 Mo, la table des pages a une taille fixe de 8 Mo (dans le pire des cas) et la table des descripteurs des pages physiques du cluster est de 56 Mo.

Pour faciliter la description des solutions proposées, nous nous placerons désormais à l'échelle d'un cluster.

2.3.3 Solutions envisagées

La première solution envisageable était d'augmenter la part de l'espace virtuel accordée au noyau. Comme celle-ci ne peut pas dépasser 4 Go (taille de l'espace virtuel entier), cela ne résolvait pas le problème du passage à l'échelle ni même le cas de notre exemple, à cause de la taille très importante de la table des descripteurs de page.

Une autre possibilité consistait à utiliser un mécanisme de *mapping* temporaire[5]. Ceci permettait

5. En supposant que l'on parvienne à éviter les différents problèmes de *deadlock* qui peuvent survenir entre les verrous

de résoudre notre problème mais présentait les inconvénients de ralentir (potentiellement de manière significative) le système d'exploitation et de ne pas exploiter l'importante mémoire physique locale au cluster qui n'est utilisée quasiment que pour projeter l'espace virtuel utilisateur.

2.4 Démarche suivie

Pour résoudre ces différents problèmes, nous avons finalement procédé selon les étapes suivantes :

- Nous avons, dès le démarrage, divisé l'espace virtuel entre les clusters comme expliqué au paragraphe 2.2.2. Le résultat de division est illustré sur la Figure 1.2 dans la partie de l'image représentant l'espace virtuel à la fin de la phase d'initialisation.
- Puis, dans chaque cluster, nous avons figé les portions de l'espace physique respectivement réservées à l'espace physique utilisateur et à l'espace physique noyau (rappelons que le sens que nous donnons à ces termes est précisé en 2.2.1). Cela implique qu'il nous a alors fallu supprimer la correspondance entre espace physique et adresses de la partie noyau de l'espace virtuel, comme expliqué en 2.2.1. Pour gérer séparément ces deux espaces, nous avons dupliqué le tableaux de pointeurs `free_pages`. Une représentation de la correspondance entre espace virtuel et espace physique découlant de l'implémentation de cette solution est donnée à la Figure 2.2.
- Enfin, nous avons sorti les tables des pages et la table des descripteurs de page de l'espace virtuel pour les gérer par un adressage physique direct. Cela signifie :
  - Pour la table des pages : nous avons supprimé « l'image virtuelle » des tables des pages initialement stockées dans le noyau et nous avons implémenté toutes les manipulations du tableau `pgdir` en adressage physique (à l'aide du registre d'extension d'adresse : le registre $24 du coprocesseur 2).
  - Pour la table des descripteurs de page physique : il s'agissait, dans chaque cluster, de supprimer la table des descripteurs des pages physiques du cluster contenue dans l'espace virtuel noyau. Il a donc fallu faire en sorte que les tableaux `free_pages` contiennent des adresses physiques - ce qui n'empêchait pas de garder des adresses de 32 bits puisque l'on ne gère que des pages du cluster courant dans un `ppm` donné (on connaît alors l'extension à rajouter par défaut aux adresses de 32 bits désignées par les pointeurs contenus dans `ppm`).

permettant de sécuriser l'accès à une page virtuelle temporairement attribuée dans une fonction donnée.

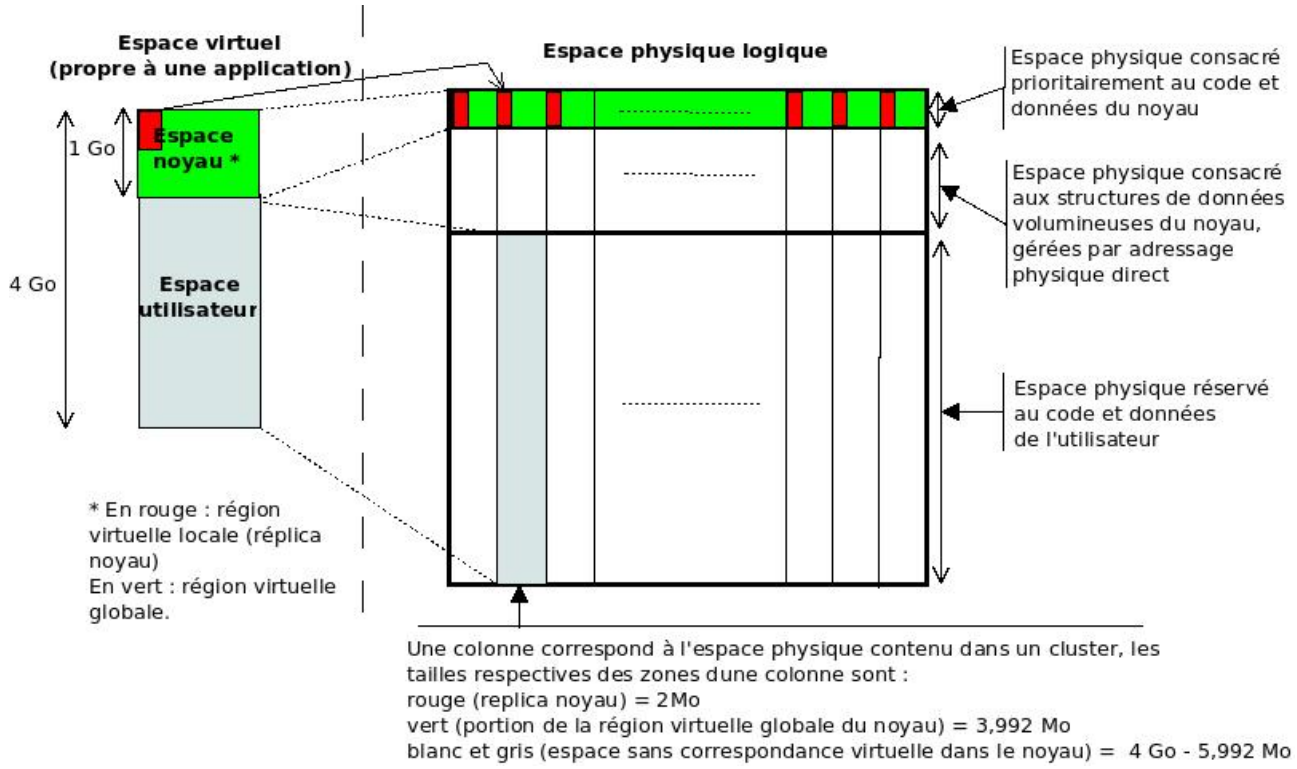

FIGURE 2.2 – Représentation de la correspondance entre espace physique et espace virtuel. Les valeurs numériques indiquées sont fournies à titre d'exemple, elles correspondent à une puce de 256 clusters contenant chacun une mémoire vive de 4 Go. Ainsi, la portion de l'espace virtuel consacrée au noyau est de 1 Go (totalité de l'espace en vert plus un espace rouge (un réplica)) et l'espace physique noyau total a une taille de $256 * 2Mo + (1Go - 2Mo) \simeq 1,498\,Go$. Notons que la séparation entre les espaces gris et blanc dans une colonne est fictive, ces espaces sont en réalité confondus.

Partie 3

Implémentation des solutions retenues

Il s'agissait de réaliser une série de fonctions implémentant la solution présentée en 2.4, tout en garantissant un fonctionnement possible (mais non optimal) si le matériel ne fourni pas la possibilité d'étendre les adresses de 32 bits à l'aide du registre d'extension présent dans les machines TSAR. Notons que nous n'avons pas eu le temps d'implémenter les mécanismes de représentation virtuelle temporaire permettant de gérer ce deuxième cas de figure.

3.1 Considérations générales

On entend par « adressage physique » l'utilisation du processeur avec la MMU désactivée et l'exploitation du registre d'extension d'adresse pour étendre les adresses de 32 bits en adresses de 40 bits. On considère qu'un passage en adressage physique est atomique si les instructions énumérées ci-dessous encadrent un faible nombre d'instructions (une seule en général). Le coût du passage en adressage physique direct et du retour en adressage virtuel est de quatre à six instructions : six si l'on souhaite garantir que la fonction puisse être appelée quelque soit le contexte d'exécution de la fonction appelante (MMU activée ou non) et quatre si l'on fait hypothèse d'un contexte d'exécution donné pour la fonction appelante. Ces instructions sont les suivantes (on pourra se référer à l'Annexe A pour un exemple de fonction implémentant un passage atomique en adressage physique) :

- Sauvegarde du registre d'état de la MMU et/ou du registre d'extension d'adresse,
- Désactivation de la MMU. [1]
- Écriture de la nouvelle extension d'adresse dans le registre $24 du coprocesseur 2.

1. Il n'est pas nécessaire de désactiver les interruptions, car nous avons modifié la routine d'interruption (la fonction exécutée à chaque fois qu'une interruption quelconque est reçue) pour qu'elle sauvegarde l'état de la MMU et le registre d'extension avant d'exécuter l'ISR et les rétablissent une fois l'interruption traitée.

- Restauration des registres sauvegardés (il faut restaurer le contexte d'exécution de la fonction appelante).

Il nous a fallu déterminer s'il était préférable d'encapsuler des fonctions entières pour les exécuter dans un contexte d'adressage physique ou s'il vallait mieux n'utiliser que de manière atomique l'adressage physique direct. Dans le premier cas, on aurait appelé une fonction de changement de mode d'adressage avant et après la série d'instructions à exécuter. Dans le deuxième cas, on ferait appel à une fonction d'écriture ou de lecture physique (en code assembleur) pour accéder directement à un mot désigné par une adresse physique donnée. Notons que la première méthode est celle qui aurait demandé *a priori* le moins de changement puisqu'il est possible de continuer à utiliser des pointeurs de 32 bits pour désigner des adresses physiques de 40 bits qui seront étendues grâce à l'extension d'un octet du côté des bits de poids forts (ces bits seront ceux qui identifient le cluster la plupart du temps). Elle impliquait cependant qu'il y aurait eu davantage d'instructions exécutées en contexte physique, ce qui pouvait s'avérer problématique pour des raisons de portabilité (les fonctions encapsulée seraient bien plus nombreuses que les deux seules fonctions d'écriture et de lecture utilisées dans le cas de la solution atomique). Nous avons donc favorisé l'emploi de la seconde possibilité.

3.2 Séparation des différents domaines de la mémoire physique

Nous avons séparé l'espace mémoire de chaque cluster en deux domaines (se référer à la Figure 2.2) : l'espace physique noyau (cf 2.2.1), dont le replica noyau fait partie, et l'espace physique non représenté dans l'espace virtuel du noyau, composé des pages allouées à l'utilisateur et de celles qui contiennent désormais les structures du noyau de tailles importantes directement gérées par adressage physique. Ces deux domaines sont gérés de manière disjointe, il nous a donc fallu contrôler dans quel domaine chaque page physique est attribuée. Pour ce faire, on a notamment créé un nouveau drapeau dans les requêtes du gestionnaire de mémoire noyau (`kmem`). Le gestionnaire de mémoire noyau (`kmem`) transmet ce drapeau aux fonctions du gestionnaire de page physique (`ppm`) qui adapte alors son comportement. En particulier, ces fonctions utilisent, selon les cas, l'un des deux tableaux des pointeurs vers les pages libres des deux domaines : une demande d'allocation dans l'espace noyau est satisfaite par l'allocation d'une page dans cet espace ou échoue tandis qu'une demande d'allocation dans l'espace physique non représenté dans le noyau tente une allocation dans cet espace, puis, en cas d'échec, dans l'espace physique noyau. Les limites de ces domaines ont été fixées à l'initialisation du système d'exploitation dans chaque cluster (donc dans

chaque `ppm`) mais peuvent éventuellement varier d'un cluster à l'autre [2].

3.3 Les tables des pages

Nous n'utilisons plus que la valeur de `pgdir_ppn`, qui indique l'adresse physique de la première case tableau `pgdir`, pour gérer la table de premier niveau des tables des pages (les tables des pages sont alignées sur des petites pages). Par deux accès physiques successifs, nous pouvons ainsi lire ou écrire la valeur de n'importe quelle entrée de la table de premier ou de deuxième niveau.

À chaque fois qu'une table est créée (`ppm_init`), copiée (`ppm_dup`) [3] , ou détruite (`ppm_release`) [4], l'utilisation d'un adressage physique atomique (qui implique qu'il faut utiliser six fois plus d'instructions que nécessaire) peut s'avérer très coûteux. Il nous a donc fallu implémenter une fonction de copie en adressage physique d'une portion de la mémoire physique vers un autre secteur de la mémoire physique qui n'est pas forcément sur le même cluster. Cela a permis d'accélérer l'initialisation et la copie mais n'a pas résolu le problème de la destruction de la table qui demande un grand nombre de lectures et écritures dans la table de premier niveau et des opérations nombreuses dans la table des descripteurs pour supprimer les tables de deuxième niveau (se référer au paragraphe 3.4 pour cette partie du problème).

3.4 La table des descripteurs de page

Il est important de noter que le nombre d'accès aux descripteurs est relativement peu élevé lors d'une opération du noyau sur la table des descripteurs de page. Il y a en effet au plus de l'ordre de 20 (deux fois le nombre d'ordres gérés) accès lors des allocations et de l'ordre de 10 accès lors de la libération d'une page. L'initialisation de la table des descripteurs fait exception puisque tous les descripteurs de tout l'espace physique à gérer dans le cluster y sont créés (cette initialisation est faite dans chaque cluster, par un des processeurs du cluster). Il en va de même pour la suppression de la table des pages puisqu'elle demande éventuellement la libération de quelques dizaines de milliers de pages dans les cas défavorables.

La manipulation de la table des descripteurs en adressage physique peut donc se faire de manière atomique en fonctionnement courant mais doit être implémentée de manière non atomique pour l'initialisation (au moins).

2. Ceci peut s'avérer utile s'il y a un grand nombre de clusters I/O et/ou de périphériques dans des clusters sans processeur. En effet, on peut vouloir augmenter la portion de l'espace physique noyau qui va être accordée au cluster en charge d'initialiser ces périphériques et de stocker les structures les identifiants.

3. Seule la table de premier niveau est copiée, celle de second niveau étant dispersée dans des petites pages allouées à la demande.

4. La destruction d'une table consiste à effacer la table de premier niveau et à libérer les pages occupées par la table de second niveau qui ne sont plus désignées par aucune autre table de premier niveau.

L'initialisation comporte deux phases déjà coûteuses en temps qui engendrent de nombreux adressages physiques : la création des pages avec l'initialisation des champs et la phase de libération de toutes les pages non réservées (c'est à dire quasiment toutes les pages) qui permet la construction des listes de pages libres. La création des pages a été accélérée en utilisant la fonction de copie en adressage physique (implémentée pour les besoins spécifiés au paragraphe précédent). La libération des pages peut difficilement être accélérée à cause de la phase d'initialisation des listes de descripteurs. Nous n'avons pas encore trouvé de solution satisfaisante à ce problème qui est en fait le même que celui de la libération massive de page lors de la suppression d'une table des pages ou de toute autre structure de taille importante. Une solution envisageable consiste à créer une représentation temporaire de la zone où se trouve une partie des descripteurs de page concernés.

Cette méthode de représentation temporaire sera, par ailleurs, la solution implémentée pour permettre la portabilité d'ALMOS sur des architectures ne présentant pas la possibilité d'étendre les adresses à l'aide d'un registre d'extension.

Conclusion

Dans ce rapport, nous avons tenté de donner un aperçu des principales modifications que nous avons dû apporter au système d'exploitation ALMOS pour lui permettre de fonctionner sur des machines dont les adresses physiques sont codées sur 40 bits et pour qu'il exploite davantage la très importante mémoire vive à sa disposition dans les machines many-cœurs sur lesquelles il est appelé à s'exécuter. Nous comptons exploiter la fin du stage (dans un mois) pour terminer le passage en adressage physique direct de la gestion de la table des descripteurs de page physique et pour tester l'impact de nos modifications sur les performances du système d'exploitation. Au terme de notre stage, nous espérons que nous aurons définitivement validé :

- la compatibilité avec le pré-loader générique,
- le portage d'ALMOS sur des puces dont les adresses physiques sont codées sur 40 bits,
- le portage d'ALMOS sur des puces utilisant des clusters sans processeurs,
- le portage d'ALMOS sur des puces ayant jusqu'à 1 To de mémoire vive,
- l'exploitation de la grande mémoire vive des puces many-cœurs par le déplacement de certaines structures de tailles importantes du noyau hors de l'espace physique représenté dans le noyau.

Ces deux deniers points ne seront atteints que pour des puces possédant un moyen d'adresser physiquement n'importe quelle case mémoire.

Deux prolongements de ce stage sont envisageables. Le premier consisterait à permettre à ALMOS de pouvoir s'exécuter sur tous les types de puces ayant davantage que 2 Go de mémoire vive (actuellement, pour fonctionner sur des puces dépourvues de moyen d'adresser physiquement toute case mémoire, ALMOS doit ignorer la mémoire dépassant 2 Go). Le second serait de gérer en adressage physique direct d'autres structures de tailles importantes comme le système de fichier par exemple.

Bibliographie

[1] Advanced locality management operating system. project home page. `https://www.almos.fr/trac/almos`, 2014.

[2] Tera-scale ARchitecture. project home page (lip6). `https://www-soc.lip6.fr/trac/tsar/`, 2014.

[3] Ghassan Almaless. *Operating System Design and Implementation for Single-Chip cc-NUMA Many-Core*. PhD thesis, Université Pierre et Marie Curie, 2014.

[4] Ghassan Almaless and Franck Wajsburt. Almos : Advanced locality management operating system for cc-numa many-cores. In *Proceedings of the 5th national seminar of GDR SoC-SIP*, Lyon, France, 2011.

[5] Ghassan Almaless and Franck Wajsburt. Does shared-memory, highly multi-threaded, single-application scale on many-cores ? In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2012.

[6] Ghassan Almaless and Franck Wajsburt. On the scalability of image and signal processing parallel applications on emerging cc-numa many-cores. In *Proceedings of the international conference on the Design and Architectures for Signal and Image Processing*, Karlsruhe, Germany, 2012. IEEE.

[7] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel (third edition)*. O'REILLY, 2005.

[8] Sangmin Seo, Junghyun Kim, and Jaejin Lee. SFMalloc : A lock-free and mostly synchronization-free dynamic memory allocator for manycores. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2011.

Annexe A

Exemple de fonction utilisant l'adressage physique direct

La fonction suivante est une fonction d'écriture en adressage physique. Les commentaires décrivent les différentes étapes.

```
static inline void cpu_phy_sw(uint64_t dest, uint32_t word)
{
  uint32_t lsb = (uint32_t)dest;
  uint32_t msb = (uint32_t)(dest >> 32);
  asm volatile(
  "mfc2   $8,     $24            \n" /* Sauvegarde du registre d'extension       */
  "mfc2   $2,     $1             \n" /* Sauvegarde du registre de mode de la MMU */
  "andi   $3,     $2,     0xb    \n"
  "mtc2   $3,     $1             \n" /* Désactivation de la MMU pour les instructions */
  "mtc2   %2,     $24            \n" /* Le registre d'extension reçoit msb       */
  "sw     %0,     0(%1)          \n" /* Écriture à l'adresse physique dest       */
  "mtc2   $8,     $24            \n" /* Le registre d'extension est restauré     */
  "mtc2   $2,     $1             \n" /* Le registre de mode de la MMU est restauré */
   :: "r" (word), "r" (lsb), "r" (msb)
   :"$2", "$3", "$8");
 }
```