Conception d'un système d'exploitation supportant nativement les architectures multiprocesseurs hétérogènes à mémoire partagée

Alexandre Bécoulet

THESE DE DOCTORAT DE
L'UNIVERSITE PIERRE ET MARIE CURIE - PARIS VI

Spécialité Informatique

Présentée par ALEXANDRE BÉCOULET

Pour obtenir le grade de
Docteur de l'Université Paris VI

CONCEPTION D'UN SYSTÈME D'EXPLOITATION SUPPORTANT NATIVEMENT LES ARCHITECTURES MULTIPROCESSEURS HÉTÉROGÈNES À MÉMOIRE PARTAGÉE.

Soutenue le : 28 septembre 2010

Devant le jury composé de :

M.	PAUL FEAUTRIER	Rapporteur
M.	FRÉDÉRIC PÉTROT	Rapporteur
M.	FRANÇOIS CHAROT	Examinateur
M.	AKIM DEMAILLE	Examinateur
M.	RENAUD PACALET	Examinateur
M.	PIERRE SENS	Examinateur
M.	ALAIN GREINER	Directeur de thèse
M.	FRANCK WAJSBURT	Encadrant de thèse

Résumé

Cette thèse présente le système d'exploitation *MutekH*, capable de s'exécuter nativement sur une plateforme matérielle multiprocesseur, où les processeurs peuvent être de complexité différente et disposer de spécificités ou de jeux d'instructions différents.

Les travaux présentés ici s'insèrent dans un contexte où les systèmes multi-core et les processeurs spécialisés permettent tous deux de réduire la consommation énergétique et d'optimiser les performances dans les systèmes embarqués et dans les systèmes sur puce.

Les autres solutions logicielles existantes permettant l'exécution d'applications sur des plateformes multiprocesseurs hétérogènes ne permettent pas, à ce jour, la communication par mémoire partagée, telle qu'on l'envisage habituellement pour les systèmes multiprocesseurs homogènes. Cette solution est la seule qui permet la réutilisation du code source d'applications parallèles existantes pour leur exécution directe par des processeurs différents.

La solution proposée est mise en œuvre en deux phases : grâce au développement d'un noyau dont l'abstraction rend transparente l'hétérogénéité des processeurs, puis à la réalisation d'un outil spécifique d'édition des liens, capable d'harmoniser le code et les données des différents fichiers exécutables chargés en mémoire partagée.

Les résultats obtenus montrent que *MutekH* permet l'exécution d'applications préexistantes utilisant des services standards, tels que les Threads POSIX, sur des plateformes multiprocesseurs hétérogènes sans dégradation des performances par rapport aux autres systèmes d'exploitation opérant sur des plateformes multiprocesseurs classiques.

Abstract

This thesis presents the *MutekH* operating system. Its kernel can run natively on a hardware platform with processors of different types in shared memory. Processors can have various complexities and different instruction sets.

This kernel can then takes advantages of both multi-cores and specialized processors based platforms at the same time. This allow to further reduce power consumption and improve performances in embedded systems and in Systems On Chip.

Other software approaches targeting heterogeneous multiprocessor support do not allow native execution in shared memory in the way common operating systems run on legacy multiprocessor platforms. *MutekH* is the only known operating system which allows reuse of existing applications source code for direct execution on different kinds of processors at the same time.

The work is divided in two parts : We first introduce the operating system kernel which is able to mask processors differences thanks to its hardware abstraction layer. We then introduce a specific heterogeneous link editor tool which allows binary files uniformization to ensure code and data can be shared by different processors once loaded in shared memory.

Results show that native heterogeneity support have no performance overhead when compared to other well known operating systems running on legacy multiprocessors platforms.

Remerciements

Je souhaite exprimer ma reconnaissance envers le professeur Alain Greiner pour sa disponibilité et les précieux conseils dont j'ai bénéficié tout au long de ma thèse. Merci également à Franck Wajsburt, mon encadrant, et à François Pêcheux qui ont suscité mon intérêt pour la recherche académique et sans qui je n'aurais probablement pas envisagé cette thèse.

Je remercie également toute l'équipe du laboratoire Lip6/SoC ainsi que les doctorants pour l'ambiance chaleureuse de travail.

J'adresse mes remerciements à Paul Feautrier et Frédéric Pétrot pour avoir accepté d'être les rapporteurs de mon manuscrit et pour le temps qu'ils ont consacré à sa relecture. Je remercie également François Charot, Akim Demaille, Renaud Pacalet et Pierre Sens pour avoir accepté d'être mes examinateurs.

Je tiens à remercier toutes les personnes qui m'ont fait l'honneur d'adopter *MutekH*, objet de cette thèse, comme système d'exploitation pour leurs travaux, notamment : Eric Guthmuller, Alain Greiner, Daniela Genius, Etienne Faure au Lip6, Fabien Colas-Bigey chez Thales et Sébastien Cerdan à Telecom ParisTech, ainsi que tous les utilisateurs présents et futurs, pour leur patience et pour tous les rapports de bogues.

Je souhaite remercier particulièrement ceux qui participent largement au développement et qui ont fait de *MutekH* leur projet : Nicolas Pouillon, Jöel Porquet, Dimitri Refauvelet au Lip6 et François Charot à l'IRISA. Les stages de Matthieu Bucchianeri et Sylvain Leroy ont également permis de faire de *MutekH* un logiciel libre de qualité.

Je remercie tout ceux qui ont contribué à la qualité de ce manuscrit, par leur relecture spontanée, leurs nombreuses remarques et corrections.

Je remercie également ma famille pour m'avoir soutenu durant mes études et particulièrement au cours de cette thèse.

Sommaire

Table des figures

Introduction

Les processeurs, dont l'évolution a été caractérisée par une augmentation continue de la fréquence de fonctionnement, suivent depuis quelques années une nouvelle voie, celle du multi-coeur. La dissipation thermique et l'énergie consommée par les microprocesseurs n'ont pas cessé de croître avec cette course à la fréquence. Les limites acceptables étant atteintes, la multiplication des processeurs dans un même système offre une autre possibilité d'augmentation de la puissance de calcul.

Afin d'exploiter ces nouvelles plateformes matérielles *multi-cores*, le logiciel doit s'adapter jusque dans le choix des algorithmes et des méthodes de développement. La démocratisation des systèmes multiprocesseurs, tout d'abord dans les ordinateurs, puis dans l'embarqué, marque en effet un tournant dans l'histoire de l'informatique.

Pour maximiser le rapport entre la puissance de calcul et l'énergie électrique consommée, l'utilisation de processeurs spécialisés pour une certaine classe d'algorithmes, est une approche classique. Elle est, par exemple, mise en place dans de nombreuses solutions embarquées exploitant des *DSP* (processeurs de signal), et elle permet également d'augmenter les performances des ordinateurs de bureau incorporant des *GPU* (processeurs graphiques).

Ces deux techniques peuvent être combinées pour concevoir des systèmes embarquant plusieurs processeurs différents et spécialisés. Il est ainsi possible de réduire encore la consommation d'énergie, tout en offrant une large gamme de services avec des performances et une autonomie accrue pour les systèmes sur batterie. Il n'est donc pas étonnant que ces systèmes multiprocesseurs hétérogènes soient de plus en plus courants, notamment dans les systèmes sur puce. Les systèmes hétérogènes haute performance pour les jeux ou les calculs scientifiques ne sont cependant pas en reste et font l'objet de divers projets de recherche.

Bien que leur démocratisation soit récente, les systèmes multiprocesseurs à mémoire partagée sont depuis longtemps utilisés dans les milieux professionnels où la puissance de calcul est un besoin critique. Le support de ces plateformes homogènes dites *SMP* (*Symetric Multi Processor*) par les systèmes d'exploitation et par les environnements de développement, est donc bien établi. Par ailleurs, l'avènement du multi-coeur dans le domaine des PC favorise la parallélisation de nombreuses applications existantes afin de tirer parti de ce type d'architecture.

Si les systèmes d'exploitation ne manquent pas, les plateformes multiprocesseurs hétérogènes ne disposent pas encore de solutions largement répandues permettant leur prise en charge. Les quelques solutions existantes se heurtent aux différences entre les processeurs, obstacles à leur coopération dans un même système, et proposent des approches dédiées et spécifiques.

Le travail de réécriture des applications, pour leur parallélisation, est déjà le principal obstacle pour l'exploitation des plateformes multiprocesseurs. Une seconde réécriture pour l'exploitation de plateformes multiprocesseurs hétérogènes n'est pas envisageable.

Le noyau du système d'exploitation *MutekH*, objet de cette thèse, tente d'apporter la solution au problème en étant le premier à supporter nativement les plateformes multiprocesseurs hétérogènes. Le support natif implique une abstraction qui rend l'hétérogénéité transparente pour l'application, en proposant des services de parallélisation standards, habituellement utilisés sur les plateformes multiprocesseurs classiques. Ceci permet la réutilisation du code des applications existantes et assure une flexibilité maximale de développement.

Le premier chapitre présente les différentes classes de plateformes hétérogènes, ainsi que les types de processeurs qu'elles embarquent. L'accent est mis sur le rôle du système d'exploitation dans le multiplexage et la gestion des ressources dans un système exécutant plusieurs tâches en parallèle. On analyse les différences entre processeurs dans une plateforme hétérogène, lesquelles sont autant de contraintes qu'il faut résoudre pour proposer un support natif de l'hétérogénéité.

Le second chapitre présente les diverses approches existantes qui tentent d'apporter une solution à l'exploitation de plateformes multiprocesseurs hétérogènes. Parmi ces solutions, l'abstraction de l'hétérogénéité des processeurs est introduite à divers niveaux : Ces techniques vont des canaux de communication spécifiques et imposés à l'application, aux langages de programmation dédiés à l'hétérogénéité, en passant par la mise en place de machines virtuelles spécialisées. La gestion de l'hétérogénéité par les systèmes d'exploitation est également évoquée, pour finalement donner un aperçu des différentes architectures logicielles courantes de noyaux qui pourraient convenir à une gestion native de l'hétérogénéité.

Le chapitre 3 établit tout d'abord les caractéristiques des plateformes hétérogènes possédant un réel intérêt, pour en déduire les contraintes liées à l'hétérogénéité qui méritent une solution efficace. Il apparaît rapidement que la couche d'abstraction du matériel joue un rôle déterminant dans la prise en charge des plateformes hétérogènes ; il convient donc d'établir une liste précise des services attendus et d'étudier les rapports de chacun à l'hétérogénéité. Une autre contrainte majeure est la génération des fichiers binaires ciblant chaque type de processeur en présence. Le compilateur, pour un processeur particulier, ignorant tout de la coopération future du code généré avec celui des autres types de processeurs, il faut un post-traitement, pour ajuster et uniformiser les fichiers binaires générés.

Les chapitres 4 et 5 détaillent respectivement l'implémentation du noyau et l'outil d'édition des liens hétérogène qui ont été développés dans le cadre de cette thèse. Un nombre considérable de problèmes techniques intéressants ont dû trouver une solution élégante pour mener à bien le projet et sont traités de façon précise. La modularité du noyau et les nombreux services disponibles sont mis en avant. L'intérêt de chaque service de parallélisation disponible est discuté et leur intérêt vis-à-vis de l'hétérogénéité est étudié.

Le dernier chapitre propose une série de mesures expérimentales dont l'enchaînement permet d'évaluer le coût en performances des solutions mises en place dans *MutekH*, pour des plateformes monoprocesseurs, multiprocesseurs classiques et multiprocesseurs hétérogènes. On démontre la portabilité du noyau sur différents types de processeurs puis on présente enfin, différentes utilisations de *MutekH*, puisqu'il est utilisé dans divers laboratoires, pour des projets sans rapport avec l'hétérogénéité.

Le développement du système d'exploitation est poursuivi quotidiennement par plusieurs contributeurs, *MutekH* étant publié en tant que logiciel libre. Un site web contenant une documentation de l'API du noyau ainsi que plusieurs tutoriels et documents de présentation du projet, est disponible à l'adresse `http://www.mutekh.org`. Ainsi nous n'avons pas jugé opportun d'inclure en annexe de cette thèse une documentation très volumineuse, plus de 700 pages, et en constante évolution.

Chapitre 1

Problématique

Les contraintes technologiques actuelles favorisent la conception de plateformes comportant plusieurs processeurs dans une même machine (multiprocesseurs) et plus récemment plusieurs processeurs dans une même puce (*multi-core*). Ces architectures sont de plus en plus répandues, aussi bien dans le domaine des processeurs généralistes haute performance, présents dans les PC, que dans les systèmes sur puce (*SoC*) rencontrés dans le monde des applications embarquées.

Si le fait d'intégrer plusieurs processeurs identiques dans une même plateforme est une pratique répandue aujourd'hui, l'utilisation de processeurs hétérogènes reste une pratique relativement peu courante car elle soulève un certain nombre de problèmes non triviaux pour le développement logiciel, lesquels ne sont généralement pas résolus de manière méthodique ou automatisée.

Le travail réalisé au cours de cette thèse propose une solution pour faciliter le développement et la réutilisation d'applications logicielles multitâches s'exécutant sur des plateformes multiprocesseurs hétérogènes.

1.1 Intérêt des systèmes multiprocesseurs hétérogènes

Même si les plateformes matérielles multiprocesseurs hétérogènes ne bénéficient pas encore du support logiciel qu'elles méritent, elles trouvent aujourd'hui leur place dans le domaine de l'embarqué et plus récemment également dans le domaine des ordinateurs classiques.

1.1.1 Processeurs spécialisés et processeurs généralistes

On peut distinguer les processeurs généralistes qui exécutent des programmes classiques, généralement écrits de manière portable dans un langage de programmation de haut niveau, et les processeurs partiellement ou complètement spécialisés qui exécutent du code écrit dans un langage spécifique. Ce développement, souvent effectué en assembleur, permet cependant d'accélérer l'exécution d'une certaine classe d'algorithmes pour lesquels ces processeurs sont conçus.

Parmi les processeurs généralistes, on peut citer les principales architectures suivantes : *x86*, *ARM*, *Mips*, *PowerPC* et *Sparc*.

Les processeurs spécialisés peuvent être de différents types ; ils possèdent généralement un jeu d'instructions orienté traitement de données, comportant des instructions *SIMD* qui opèrent le même traitement sur plusieurs données en parallèle, des instructions effectuant des opérations arithmétiques complexes en virgule fixe ou en virgule flottante, ou des instructions plus spécialisées encore. C'est le cas des processeurs de signal numérique (*DSP*) comme ceux de la famille *ST200* proposés par *STMicroelectronics* ou ceux de la famille *TMS320* proposés par *Texas Instruments*.

La frontière entre ces deux classes de processeurs devient parfois floue avec l'apparition des extensions du jeu d'instructions des processeurs généralistes. Les opérations en virgule flottante sont aujourd'hui intégrées à part entière dans les processeurs haute performance et des extensions diverses se développent également ; ainsi à titre d'exemples pour les architectures citées :

– *x86* : la série d'extensions *MMX* fournit des opérations *SIMD* entières sur des mots de 64 bits et la série *SSE*, des opérations *SIMD* flottantes sur des mots de 128 bits.
– *ARM* : l'extension *NEON*[7] apporte également des opérations *SIMD* entières et flottantes sur des mots de 128 bits.
– *Mips* : *MDMX* complète le jeu d'instructions entières, *MIPS-3d* accélère quelques opérations courantes des moteurs 3d, et plus récemment l'extension *MIPS DSP*[6] propose des opérations *SIMD*.
– *PowerPC* : l'extension *Altivec*[2] apporte des opérations *SIMD* entières et flottantes sur des mots de 128 bits.
– *Sparc v9* : l'extension *VIS* permet de réaliser des opérations *SIMD* entières dans les registres généraux de 64 bits.

Si ces extensions sont monnaie courante dans les processeurs haute performance des PC, les processeurs généralistes destinés à l'embarqué n'intègrent pas nécessairement ces instructions complexes qui ont un coût important en surface de silicium et donc, en consommation.

Il est important de souligner également que ces jeux d'instructions spécialisés doivent souvent être utilisés de manière explicite par les développeurs. Même si certains compilateurs peuvent, dans certains cas, générer quelques instructions *SIMD* à partir de langages de haut niveau, il reste difficile d'exploiter le parallélisme d'un algorithme s'il n'est pas explicitement décrit par l'implémentation.

Cas des systèmes embarqués

Les processeurs spécialisés étant capables d'exécuter certains algorithmes plus efficacement que les processeurs généralistes, les contraintes de consommation électrique dans les systèmes embarqués poussent souvent à leur utilisation, notamment pour exécuter les algorithmes coûteux de traitement du signal, multimedia ou encore cryptographiques.

Le processeur généraliste étant par ailleurs souvent incontournable pour l'exécution du système de base, c'est dans l'embarqué que l'on rencontre le plus souvent des plateformes hétérogènes encore aujourd'hui.

Cas des plateformes haute performance

La recherche de performances maximales, dans les ordinateurs de bureaux et dans les serveurs de calcul, pousse également à l'utilisation de processeurs spécialisés en complément du processeur

principal généraliste.

On constate que les cartes graphiques des PC actuels, destinées aux applications 3d, intègrent des processeurs capables d'exécuter du code. C'est le cas des cartes graphiques à base de *GPU*s *Nvidia* ou de *GPU*s *Ati* lesquels sont programmables en C pour réaliser des opérations de calcul sur des données flottantes qui ne sont pas en rapport avec les fonctions d'affichage de la machine. Ces fonctionnalités sont aujourd'hui utilisées pour réaliser des calculs scientifiques avec un gain en performances d'un facteur de l'ordre de la dizaine par rapport au processeur central.

On peut considérer que les applications qui utilisent les technologies *CUDA*[3] de *Nvidia* ou *Close to Metal*[9] de *AMD/Ati* s'exécutent sur une plateforme hétérogène constituée par le processeur principal et le processeur graphique du PC.

1.1.2 Processeurs simples et processeurs haute performance

Les contraintes technologiques limitant une montée en fréquence des processeurs, la tendance est à l'intégration de plusieurs processeurs sur une même puce pour répondre au besoin croissant de puissance de calcul.

Cette nouvelle tendance technologique a un impact important sur le développement des applications logicielles qui doivent être programmées de manière parallèle pour tirer pleinement parti de ces nouvelles architectures.

Ceci implique un effort de développement incontournable de réécriture des algorithmes implémentés de manière non parallèle, dans les applications déjà existantes. Cependant cette adaptation n'est pas toujours possible ; en effet, on constate, qu'il existe deux classes d'algorithmes :

– Les algorithmes intrinsèquement parallélisables qui peuvent être implémentés en utilisant plusieurs *processus* ou *thread*s du système d'exploitation pour tirer parti des multiples processeurs.
– Les algorithmes intrinsèquement séquentiels qui ne sont parallélisables en aucun cas.

Dans un système embarqué dédié à une application particulière, il est possible d'exploiter à coup sûr le parallélisme de la plateforme matérielle par des mécanismes conjoints de l'architecture matérielle et de l'application logicielle.

Sur une station de travail ou un serveur, la situation est différente :

– les algorithmes séquentiels doivent s'exécuter avec un niveau de performances correct, équivalant à celui des systèmes monoprocesseurs haute performance que l'on utilise couramment
– Les algorithmes parallélisables doivent être distribués sur un grand nombre de processeurs pour réduire les temps de calcul.
– La consommation électrique des processeurs haute performance limite leur nombre au sein d'une même plateforme ou d'une même puce.

Ce constat devrait mener à l'apparition de plateformes *many-core* hétérogènes où coexistent, dans la même puce, un petit nombre de processeurs haute performance et une multitude de processeurs simples généralistes ou partiellement spécialisés.

L'architecture *Cell*[4] d'*IBM*, qui embarque un processeur *PowerPC* généraliste et 8 processeurs simples destinés au calcul et dotés d'instructions *SIMD*, peut être considérée comme précurseur de cette famille de systèmes.

D'autres projets de recherche s'intéressent aujourd'hui aux architectures *many-core* et proposent des puces intégrant des processeurs simples en nombre beaucoup plus important. C'est le cas par exemple de l'architecture *TSAR* [1], qui vise à intégrer un grand nombre de processeurs regroupés en *clusters* interconnectés par un réseau sur puce (*NoC*).

On peut concevoir deux types de plateformes hétérogènes :

– Les architectures multiprocesseurs hétérogènes où les processeurs disposent de jeux d'instructions différents et ne peuvent en aucun cas partager le même code binaire exécutable.
– Les architectures multiprocesseurs hétérogènes où les processeurs disposent du même jeu d'instructions mais n'ont pas les mêmes architectures et complexités internes. Ces processeurs peuvent éventuellement partager le même code binaire mais cette approche n'est pas toujours retenue.

1.2 Le rôle du système d'exploitation

Le système d'exploitation joue un rôle primordial dans les architectures multi-core hétérogènes car, en plus de gérer l'attribution des ressources classiques telle la mémoire, les périphériques ou le temps processeur, il doit répartir et placer les différentes tâches logicielles sur les différents processeurs.

Pour bien comprendre les enjeux du développement d'un système d'exploitation capable de s'exécuter nativement sur une plateforme multiprocesseur hétérogène, il convient d'établir la liste des services indispensables au fonctionnement d'une application parallèle sur un système multiprocesseur.

1.2.1 Abstraction du matériel

Les noyaux portables des systèmes d'exploitation modernes possèdent généralement une couche d'abstraction du matériel. Il s'agit d'une *API* interne qui propose des services d'accès spécifiques au matériel de la plateforme et du processeur. Il s'agit de la gestion des interruptions ou des accès atomiques à la mémoire par exemple.

Différentes implémentations de cette *API* sont réalisées pour supporter les différentes plateformes et architectures de processeurs visées par le système d'exploitation. Ceci permet en principe d'éviter les modifications dans les couches hautes du système lors du portage vers une nouvelle architecture. La gestion des périphériques est en général exclue de cette *API* et reste réservée aux pilotes de périphériques.

1.2.2 Parallélisation des applications

Contextes d'exécution

Le contexte d'exécution est constitué par l'état des registres utilisateurs du processeur. Il évolue donc à chaque instruction exécutée par le processeur.

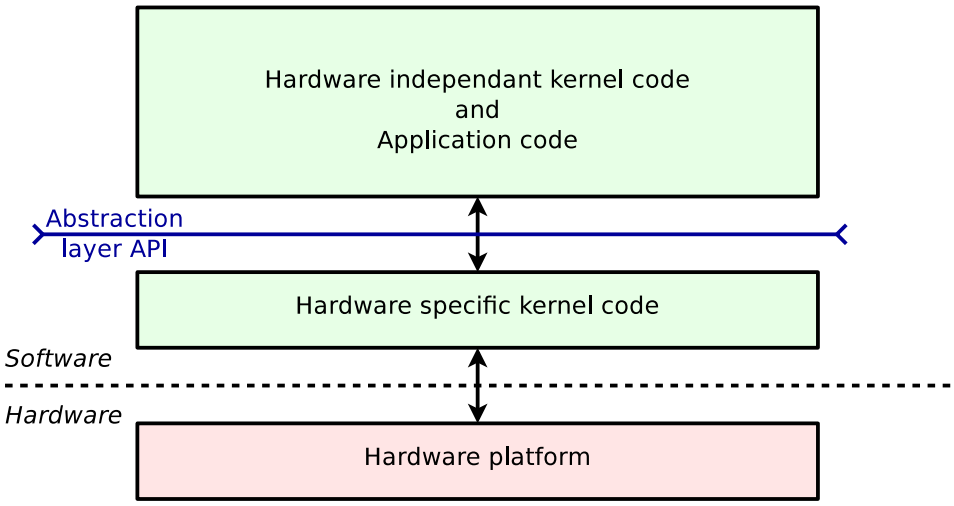

FIGURE 1.1 – Couche d'abstraction dans un noyau de système d'exploitation

Un système d'exploitation multitâche doit être capable de sauvegarder, de restaurer et de permuter le contexte en cours d'exécution d'un processeur, dans des structures en mémoire. C'est la base des objets logiciels que l'on nomme tâches, *thread*s ou *processus*.

Le format en mémoire des contextes d'exécution sauvegardés dépend directement du jeu de registres du processeur que l'on considère, et donc, de son architecture. Dans le cas d'un système hétérogène, il n'est donc pas possible de restaurer un contexte d'exécution qui a été sauvegardé sur un processeur d'un autre type.

Ordonnancement et migration

Les primitives de sauvegarde et de restauration du contexte d'exécution ne suffisent pas pour constituer la gestion du multitâche dans un système d'exploitation. Le mécanisme chargé de contrôler l'enchaînement des tranches de temps allouées aux différentes tâches qui s'exécutent sur un processeur ou sur un groupe de processeurs, est également essentiel.

L'ordonnanceur est composé de structures de données (généralement une ou plusieurs listes de tâches) et d'algorithmes plus ou moins complexes qui décident de l'ordre et de la durée d'exécution de chaque tâche.

On peut considérer que chaque tâche existante sur le système peut se trouver dans trois états :

– La tâche est en cours d'exécution par un processeur.
– La tâche est active et se trouve inscrite dans une structure de données de l'ordonnanceur, appelée file d'exécution, en attente d'élection pour être exécutée aussitôt que possible.
– La tâche est inactive ou endormie et n'est plus inscrite dans la file d'exécution pour l'instant. Dans ce cas, elle est probablement inscrite dans une des multiples files d'attente qui existent dans le système, en attendant d'être réveillée par un événement quelconque.

Dans un système multiprocesseur, il existe plusieurs possibilités d'affectation des tâches aux différents processeurs :

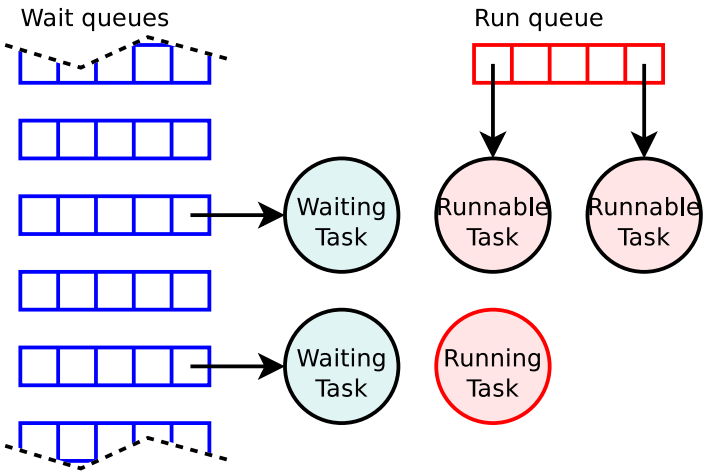

FIGURE 1.2 – Files d'attente et file d'exécution des tâches

– Il peut exister une seule et unique file d'exécution pour tous les processeurs ; dans ce cas chaque tâche peut être exécutée indifféremment par n'importe quel processeur à chaque élection (migration systématique).
– Il peut exister autant de files d'exécution que de processeurs ; dans ce cas une tâche est toujours exécutée par le même processeur, sauf si elle est déplacée explicitement (migration explicite).
– Il peut exister des implémentations hybrides où des groupes de processeurs partagent une des files d'exécution du système.

Le choix de l'organisation et du nombre de files d'exécution dépend souvent de l'architecture matérielle sous-jacente, notamment de l'organisation des caches et de la cohérence mémoire qui déterminent le coût de la migration.

Primitives de synchronisation

Si le nombre de files d'exécution est limité par le nombre de processeurs de la plateforme, les files d'attente sont des objets logiciels qui peuvent être alloués statiquement ou dynamiquement en nombre illimité.

Dans une application multitâche, où chaque tâche possède un rôle de traitement, les dépendances sur les données à traiter, qui peuvent être fournies par une autre tâche ou provenir de l'extérieur via un périphérique, déterminent si la tâche doit s'exécuter ou si elle n'a pas matière à travailler à un instant donné. Chaque tâche qui doit attendre la disponibilité d'une donnée ou d'une ressource peut s'inscrire sur une file d'attente associée.

On peut donc établir deux primitives d'ordonnancement de base :

– L'action d'inscrire sur une file d'attente quelconque, préalablement allouée, une tâche qui s'exécute actuellement et de permuter le contexte d'exécution du processeur sans la remettre sur la file d'exécution (endormissement).
– L'action de retirer une tâche d'une file d'attente pour la remettre dans la file d'exécution qui lui est ordinairement affectée (réveil), de telle sorte qu'elle puisse être à nouveau élue pour l'exécution,

à l'occasion d'un prochain changement de contexte.

Ces deux primitives permettent de construire toutes les primitives de synchronisation de haut niveau que l'on rencontre dans les différentes interfaces logicielles des bibliothèques et noyaux des systèmes d'exploitation. On peut citer, par exemple, les *verrou*s, *sémaphore*s et *barrières* présents dans certaines bibliothèques standards, telle la bibliothèque de *thread*s *POSIX* dite *PThread*.

Ces primitives de synchronisation s'appuient sur les opérations mémoire atomiques de la couche d'abstraction du matériel pour garantir la synchronisation des processeurs au niveau matériel.

1.2.3 Communication entre tâches

Il est indispensable que les multiples tâches d'une application parallèle ne soient pas isolées. Elles doivent communiquer selon les besoins des algorithmes répartis mis en œuvre.

Différentes méthodes de communication sont envisageables et sont répandues dans les systèmes d'exploitation courants.

Partage de la mémoire

Une technique courante pour permettre la communication entre deux tâches logicielles est le partage de l'espace d'adressage mémoire.

Les tâches peuvent partager tout ou une partie seulement de leur espace d'adressage mémoire, on les appelle respectivement *thread*s ou *processus*.

Les *thread*s ou *processus* accèdent à diverses données et variables allouées en mémoire et sont simplement synchronisés grâce aux primitives de synchronisation mentionnées précédemment. Il n'existe pas d'autres contraintes fortes sur l'implémentation ; le développement d'algorithmes répartis n'est pas réellement encadré par un service de communication inter-tâches spécifique.

Cette technique est proposée par les systèmes *UNIX*. Ceux-ci permettent le partage de mémoire entre les *processus* via l'*API* de *shared memory*. Elle est également supportée nativement par plusieurs systèmes d'exploitation qui implémentent des *thread*s ainsi qu'un jeu de primitives de synchronisation, comme c'est le cas de la bibliothèque de *thread*s *POSIX*.

Bien que cette technique soit particulièrement bien adaptée aux systèmes non scalables, et très répandue dans les machines monoprocesseurs ou multiprocesseurs telles que les PCs, il existe des projets d'architectures *many-core* à mémoire partagée [1].

Passage de messages

Une autre technique est le passage de messages entre les tâches, à travers des canaux de communication spécifiques. Cette technique implique l'utilisation d'une *API* spécifique qui prend en charge l'envoi et la réception des messages.

Elle a l'avantage de formaliser et de faire apparaître de manière explicite la communication entre les tâches, au travers d'un graphe de tâches, mais ne permet pas la réutilisation directe de code conçu pour fonctionner en mémoire partagée.

Les algorithmes répartis doivent être implémentés spécifiquement pour chaque jeu de primitives de communication. Différents jeux proposent des services variant sur plusieurs points :

– Les communications peuvent être avec ou sans connexion entre les tâches : dans le premier cas des canaux de communication sont préétablis.
– Les canaux de communication peuvent être préétablis de manière statique ou créés dynamiquement pendant l'exécution.
– Les messages peuvent être délivrés de manière synchrone ou asynchrone : dans le premier cas seulement, la tâche émettrice reste bloquée jusqu'à la prise en compte du message par l'autre tâche.
– Les messages peuvent être délivrés avec ou sans garantie d'acheminement.
– Un canal de communication peut éventuellement accepter plusieurs tâches émettrices.
– Un canal de communication peut éventuellement cibler plusieurs tâches réceptrices.

Le passage de messages convient aussi bien pour les systèmes à mémoire partagée que pour les systèmes à mémoire distribuée ; il est de ce fait utilisable dans les *SoC* mais aussi sur les clusters de machines.

On peut par exemple citer la bibliothèque de passage de messages *MPI* disponible pour un très grand nombre de systèmes.

Dépendance vis-à-vis de l'infrastructure matérielle

On peut superposer deux types d'infrastructures matérielles aux deux modes de communication logicielle que l'on vient de présenter. On aboutit à ce constat :

– Les architectures matérielles à mémoire partagée permettent d'implémenter nativement les deux modes de communication.
– Les architectures matérielles sans partage de la mémoire ne peuvent supporter nativement que les passages de messages.
– Il existe toutefois des techniques permettant d'émuler la mémoire partagée, lorsqu'elle ne l'est pas physiquement, par interception des accès mémoire et passages de messages, dans les architectures où le matériel supporte la protection de la mémoire. Cette approche implique généralement un coût important en performances et n'est pas retenue.

1.3 Les contraintes liées à l'hétérogénéité

L'hétérogénéité des processeurs dans la plateforme soulève un certain nombre de problèmes aux concepteurs de logiciel. Voici les principales divergences de caractéristiques entre les types de processeurs existants :

– Différences et spécificités des mécanismes système (instructions atomiques, interruptions...)
– Différences des jeux d'instructions.
– Différences des largeurs de types entiers et contraintes d'alignement des données.

Dans le cadre du développement d'un système d'exploitation unique, qui s'exécute sur une plateforme hétérogène à mémoire partagée, il convient d'étudier de manière détaillée ces divergences

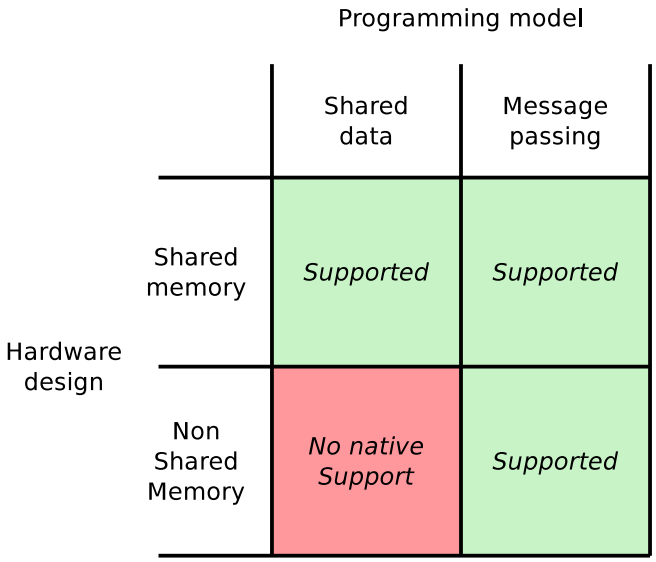

FIGURE 1.3 – Support des modes de communication selon l'infrastructure matérielle

pour mieux comprendre comment les masquer et appréhender les points communs pour développer l'abstraction nécessaire à la résolution du problème.

1.3.1 Différences des mécanismes système

Toutes les familles de microprocesseurs destinées à la réalisation de plateformes multiprocesseurs proposent des mécanismes système élémentaires comme les interruptions, les accès atomiques ou encore la virtualisation de la mémoire.

En dehors du détail d'implémentation, certaines caractéristiques des processeurs marquent leur appartenance à des grands groupes prédéterminés :

– Les processeurs *big endian* et *little endian* qui se différencient par l'ordre de stockage des octets des mots en mémoire.
– Les processeurs *CISC* et *RISC* qui se différencient par la complexité de leur jeu d'instructions et la longueur variable ou fixe de leurs *opcode*s.
– Les processeurs systèmes qui prennent en charge de manière câblée certaines opérations habituellement réservées aux systèmes d'exploitation.

Si les processeurs implémentent généralement la plupart de ces mécanismes sans trop d'originalité, il arrive que certains points les différencient radicalement. En effet, des choix de conception originaux, pour résoudre un problème particulier, augmentent encore la diversité. On peut citer entre autres :

– Les fenêtres de registres des processeurs *Sparc*.
– Les modes d'exécution et d'interruption des processeurs *ARM*.
– La gestion câblée de la permutation de contextes des processeurs *x86*.

Certains de ces mécanismes exotiques, finalement trop spécifiques, sont souvent inutilisés par les développeurs de logiciels, au profit de la portabilité du code.

Finalement, l'abstraction nécessaire à la coexistence de différents processeurs dans une même plateforme est nécessaire à plusieurs niveaux :

– Au niveau de l'interface logicielle entre les couches basses du noyau et les couches hautes indépendantes du matériel. Il s'agit de masquer les différences entre matériels supportés. On remarque que ce problème est également présent dans les systèmes homogènes classiques qui veulent être portables sur différentes plateformes et différents processeurs.
– Au niveau de l'accès à l'espace de mémoire partagée, où les différences de comportement et de format des accès aux données ne sont pas tolérables entre les processeurs hétérogènes.

Démarrage des processeurs

Les processeurs se différencient sur deux points importants quant au démarrage du système :

L'adresse de démarrage qui désigne la première instruction exécutée lorsque le processeur démarre constitue un premier problème car elle peut entraîner une collision des codes de démarrage des processeurs s'ils sont égaux ou trop proches.

L'autre point concerne le protocole de démarrage des multiples processeurs dans un système multiprocesseur. Plusieurs stratégies de conception matérielle existent :

– Un seul processeur maître démarre tout d'abord, puis il réveille les autres processeurs le moment venu, afin qu'ils démarrent à leur tour.
– Tous les processeurs démarrent en même temps et des mécanismes de synchronisation logiciels ou d'identification matériels se chargent de différencier les initialisations effectivement réalisés par les processeurs.

Un système d'exploitation hétérogène peut être amené à gérer ces deux cas dans une même plateforme matérielle. Contrairement à un noyau classique, ce problème doit être géré de manière dynamique à l'exécution et non de manière statique à la compilation, en fonction du type de processeur cible.

Gestion des interruptions

La gestion des mécanismes d'interruption, et plus précisément l'interface plus ou moins complexe qui existe entre le logiciel et le matériel, varie d'un processeur à l'autre :

– Une ou plusieurs lignes d'interruption peuvent exister
– Le déclenchement d'une interruption peut provoquer un branchement à une adresse fixe (point d'entrée). Dans d'autres cas le processeur consulte une table d'interruption en mémoire, de manière câblée, pour déterminer le point d'entrée.
– Le démultiplexage du numéro de la ligne d'interruption peut être effectué par le processeur ou laissé à la discrétion du logiciel.
– Le changement de mode d'exécution à l'entrée et à la sortie du gestionnaire d'interruption peut être câblé, partiellement câblé ou non câblé. Certains processeurs imposent des modes de protection spéciaux et une permutation du banc de registres lors des interruptions.

Accès atomiques

Les accès atomiques, indispensables pour les primitives de synchronisation, sont présents dans les différents jeux d'instructions des processeurs, sous différentes formes souvent adaptées à différents types d'interconnexions avec la mémoire :

- Les processeurs, développés à l'origine pour cohabiter avec d'autres processeurs sur un *bus*, proposent souvent des instructions de verrouillage du bus qui lui garantissent l'exclusivité pendant un cycle complet, constitué d'une lecture puis d'une écriture. C'est le cas du processeur *x86*.
- D'autres processeurs offrent des instructions permettant d'échanger atomiquement les valeurs contenues dans un registre et dans une case mémoire.
- D'autres encore proposent une manière flexible de réaliser une opération de lecture/écriture atomique grâce à une instruction dédiée de lecture avec réservation, puis d'écriture conditionnelle. C'est cette approche qui est la plus adaptée aux architectures scalables, généralement non basées sur des bus.

Les processeurs choisis doivent bien sûr partager la même interconnexion avec la mémoire et utiliser des mécanismes matériels compatibles d'atomicité ; les formats et le jeu d'opérations atomiques utilisés par le logiciel doivent toutefois être harmonisés.

1.3.2 Harmonisation des structures de données

Le système d'exploitation et l'application qui s'exécutent nativement sur une plateforme hétérogène à mémoire partagée partagent leurs structures de données en mémoire. Cela signifie que tous les processeurs de la plateforme doivent pouvoir accéder à ces structures partagées, de la même manière, quel que soit leur type.

Au niveau du code source, les structures de données sont décrites grâce au langage de programmation employé. Le format binaire de ces structures dépend du code source, mais également du compilateur utilisé, des options de compilation et des largeurs de types choisies par convention pour le processeur cible.

Le seul point qui importe réellement dans les systèmes homogènes classiques, concerne l'uniformité de la génération du code. C'est-à-dire que, si tout le code binaire est généré avec le même compilateur, aucun problème lié à diverses représentations des structures de données en mémoire, ne se pose. Il n'en va pas de même dans une plateforme hétérogène.

Largeur des types entiers

Les tailles des types entiers standards dans des langages de haut niveau, ne sont pas fixes et peuvent varier d'une architecture de processeur à l'autre. Voici quelques faits relatifs au langage C :

- Le type `long` possède la même largeur qu'un pointeur. Cette largeur dépend de la largeur des adresses manipulées par le processeur.
- Le type `int` peut contenir 16 ou 32 bits.
- Le type `char` peut être signé ou non signé.

Les programmes C utilisent ces types couramment et prennent parfois pour acquis que leurs tailles sont fixes ; ceci s'avère une cause importante de non portabilité du code.

La diversité de ces types pose un réel problème pour la cohabitation des codes d'un système hétérogène qui partage ces données en mémoire.

Alignement

L'alignement des données en mémoire doit être respecté par le compilateur alors que le développeur ne se pose pas systématiquement la question de l'alignement des champs dans une structure. Ce travail d'alignement du compilateur peut parfois être effectué de différentes manières. Si la compilation, pour les diverses architectures de processeurs visées, ne respecte pas les mêmes règles, les codes binaires générés ne seront pas capables de coopérer convenablement.

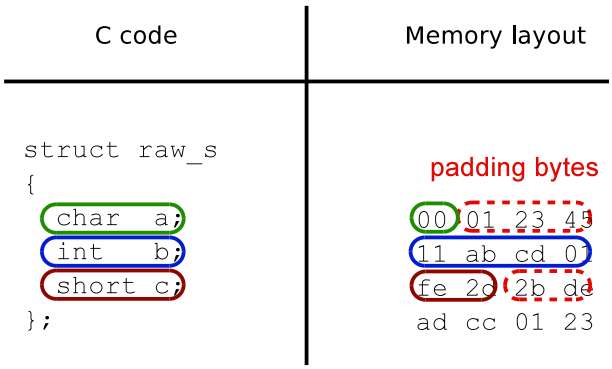

FIGURE 1.4 – Exemple d'alignement des champs d'une structure C

L'endianness

L'ordre du stockage des octets d'un mot dans la mémoire, s'avère une contrainte évidente de conception et de communication dans les systèmes à mémoire partagée avec processeurs hétérogènes. Cette contrainte apparaît déjà dans les applications qui mettent en œuvre la communication des ordinateurs en réseaux, mais également dans la conception des formats de fichiers.

Comportement de la chaîne de compilation

Si l'agencement en mémoire des éléments tels que les structures ou les tableaux est un premier point de divergence entre les compilateurs, l'ordre dans lequel les différentes variables ou fonctions sont rangées dans les sections assemblées est un second point de divergence car il ne répond à aucune règle.

En effet, le fonctionnement interne du compilateur construit et organise ces symboles en mémoire pendant la compilation, puis opère une sérialisation pour les écrire de manière contigüe dans le fichier de sortie. Il est donc probable que l'ordre dans lequel ces symboles sont sérialisés ne dépend que des structures de données et des algorithmes du compilateur, pouvant se baser sur une table de hashage ou un arbre équilibré, par exemple.

L'expérience montre que deux codes sources compilés pour deux architectures de processeurs cibles différents, avec une même version du compilateur *gcc*, donnent un ordre des *symbole*s différent et aléatoire.

Si cet ordre n'est habituellement pas important, le partage des symboles en mémoire par les différents types de processeurs du système hétérogène, nécessite que les symboles se trouvent aux mêmes adresses dans les sections, lors des compilations vers les différentes cibles

1.3.3 Incompatibilité des codes binaires

Une des contraintes les plus évidentes reste la différence des jeux d'instructions entre les processeurs. Alors que toutes les données doivent être partagées, les processeurs ne peuvent pas exécuter le même code compilé, lequel doit donc exister sous plusieurs formes en mémoire.

Toutefois, il arrive souvent qu'un symbole de donnée, comme une variable, pointe sur un symbole de code. Aussi les pointeurs sur fonctions posent un vrai problème pour la cohabitation des différents codes puisqu'une telle variable ne peut à priori pointer simultanément sur les codes des différentes fonctions compilées pour les différentes architectures de processeurs.

1.4 Conclusion

Cette thèse a pour objectif la conception d'un système d'exploitation capable de s'exécuter et de permettre à l'application de s'exécuter nativement sur une plateforme matérielle multiprocesseur à mémoire partagée, où les processeurs sont différents par leur architecture ou par leur implémentation d'une même architecture. La gestion native de l'hétérogénéité doit permettre l'utilisation d'applications existantes sans développement supplémentaire.

Les différences entre processeurs, liées à leurs jeux d'instructions et à leur conception, ne posent habituellement qu'un problème mineur d'abstraction du matériel aux concepteurs de systèmes d'exploitation. Mais lorsque ces processeurs doivent cohabiter, ces divergences sont autant d'obstacles à la réalisation d'un noyau performant, fournissant les services classiques attendus par l'application, tout en supportant nativement l'hétérogénéité de la plateforme matérielle.

Voici les questions auxquelles on s'efforcera d'apporter une réponse pertinente :

– Quelles sont les différences entre processeurs qui peuvent être masquées élégamment et quelles sont celles que l'on ne rencontrera pas dans une situation réaliste ?
– Comment apporter des solutions techniques logicielles pour contourner chaque contrainte liée à l'hétérogénéité sans pénaliser les performances ?
– Quelle architecture de noyau et quels services de communication inter-tâches peuvent convenir si l'on tente de maximiser la généricité et la réutilisabilité du code de l'application, malgré les contraintes liées à l'hétérogénéité ?
– Quelles sont les caractéristiques de la plateforme matérielle qui permettent l'exécution d'un tel système d'exploitation ?

Chapitre 2

Etat de l'art

Ce chapitre s'organise en deux parties : Notre analyse s'intéresse d'abord aux différentes approches existantes qui tentent de résoudre le problème de l'hétérogénéité des processeurs, puis elle présente ensuite les différentes architectures de noyaux de systèmes d'exploitation, candidates au développement de notre système gérant l'hétérogénéité nativement.

2.1 Gestion de l'hétérogénéité

A ce jour il n'existe pas de références à un système d'exploitation qui tente de résoudre le problème de l'hétérogénéité de manière native [11].

Il existe cependant plusieurs autres approches qui supportent les divers types de plateformes matérielles hétérogènes existants. Chacune d'elles met en place des solutions plus ou moins complexes, en misant parfois sur un aspect particulier des plateformes dont on cherche à tirer parti.

2.1.1 Canaux de communication spécifiques

L'approche la plus directe pour exploiter une plateforme multiprocesseur hétérogène est l'exécution de deux systèmes d'exploitation différents, ou de deux instances différentes du même système d'exploitation, avec l'utilisation d'un mécanisme matériel spécifique pour la communication entre les multiples sous-systèmes. Chaque système gérant généralement son propre espace d'adressage, les communications inter-systèmes peuvent être implémentées sous forme de *FIFOs* matérielles ou de convertisseurs de formats spécifiques, mais il n'est pas question ici d'utilisation de la mémoire partagée.

Sans abstraction logicielle

Cette approche ne met en place aucune abstraction logicielle (figure 2.1). Chaque système peut percevoir les autres systèmes comme un simple périphérique.

Ce genre de plateforme se rencontre dans l'embarqué où l'utilisation de processeurs hétérogènes est motivée par la diversité des services et périphériques spécifiques qui accompagnent les processeurs et *DSP* composant la plateforme.

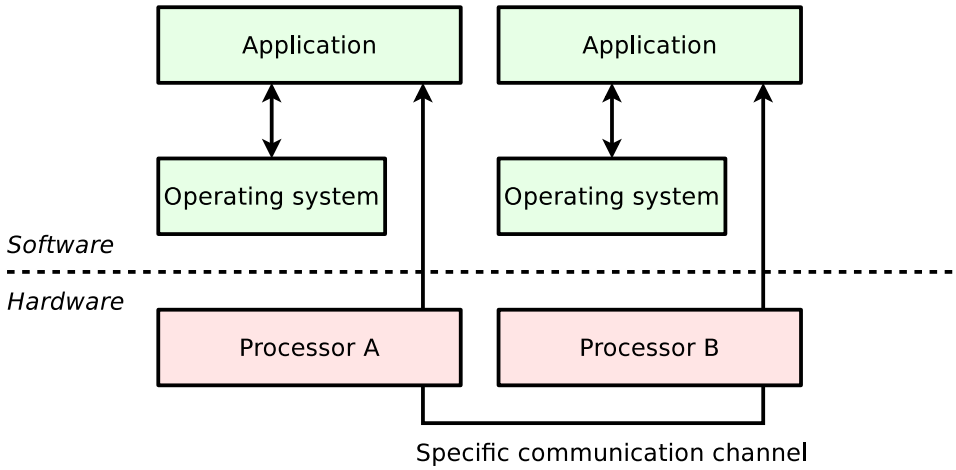

FIGURE 2.1 – Architecture simplifiée d'un système hétérogène sans abstraction

Si cette approche simple et peu élégante est souvent utilisée dans l'industrie pour résoudre ponctuellement et rapidement le problème de l'hétérogénéité, elle ne fait pas l'objet de recherches dans le monde scientifique.

Ses limites sont évidentes :

– Pas d'abstraction, le logiciel doit être développé de manière spécifique.
– Pas de flexibilité, on ne peut pas redéployer une tâche sur un processeur différent du fait de la différence logicielle des deux sous-systèmes.

Avec abstraction logicielle

Dans ce cas la gestion du canal matériel de communication n'est pas directement à la charge de l'application, une abstraction logicielle sous différentes formes peut exister :

– Utilisation d'une bibliothèque de passage de messages (figure 2.2).
– Utilisation d'un langage spécifique de description de graphe de tâches communicantes.
– Utilisation de code virtualisé.
– Utilisation de tout autre mécanisme qui masque l'implémentation matérielle des canaux de communication en imposant des restrictions et contraintes sur le développement de l'application.

Ces différentes solutions, qui peuvent également être mises en œuvre via des mécanismes en mémoire partagée, sont traitées dans la suite de ce chapitre.

2.1.2 Zones de mémoire partagée contrôlées par l'OS

L'exploitation de la mémoire partagée de la plateforme est une approche qui paraît naturelle pour le développeur d'applications. Elle pose cependant plusieurs problèmes, cités précédemment, lorsque l'on s'intéresse aux systèmes multiprocesseurs hétérogènes.

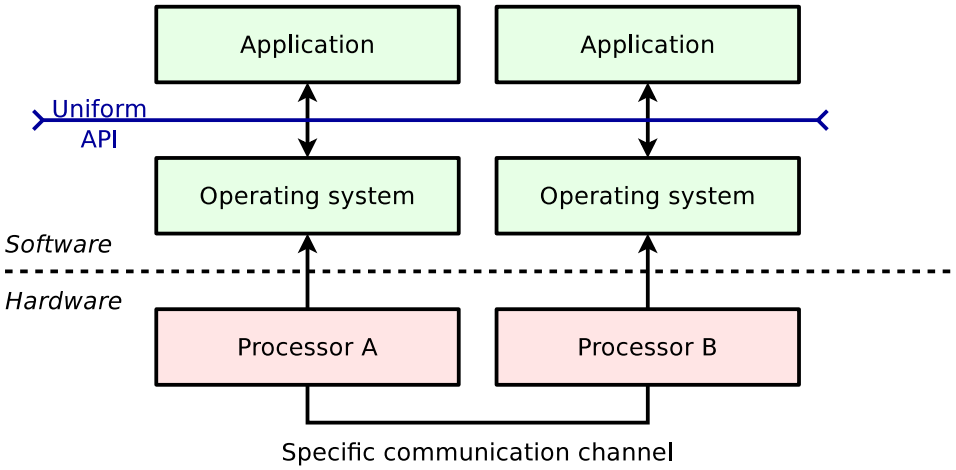

FIGURE 2.2 – Architecture simplifiée d'un système hétérogène avec abstraction

Certains noyaux de systèmes d'exploitation supportent une grande diversité de plateformes matérielles et doivent de ce fait prendre en compte et contrôler les différents mécanismes liés à la mémoire, nous pouvons citer :

– les effets de cache avec ou sans cohérence.
– les barrières de synchronisation mémoire au niveau du compilateur et du processeur.
– la pagination et la protection de la mémoire
– le placement des données en mémoire dans les architectures *NUMA*.

La mémoire partagée, dont il est question ici, est un service de communication encadré par le système d'exploitation, qui est capable d'allouer des zones de mémoire partagées entre plusieurs contextes d'exécution, à la demande des applications (figure 2.3). Les accès en eux-mêmes peuvent également être encadrés par des appels au système ou non.

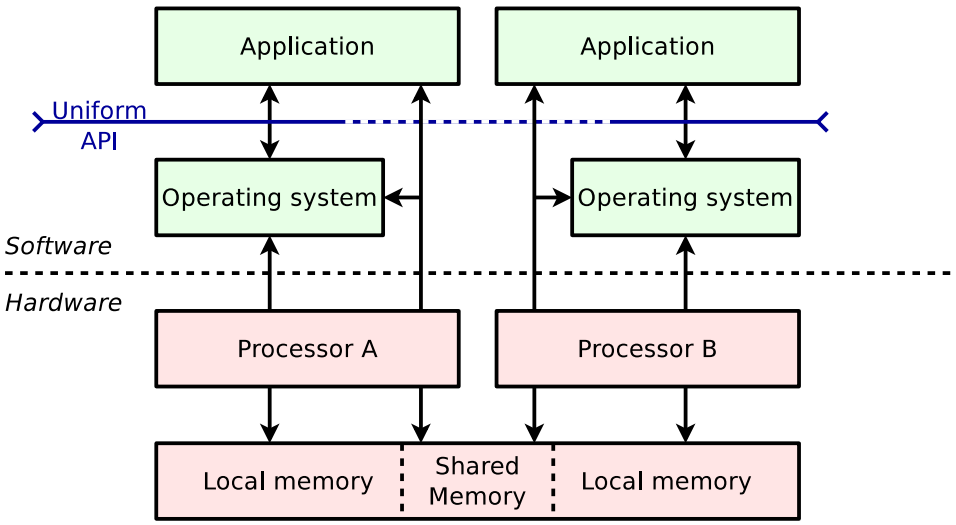

FIGURE 2.3 – Architecture simplifiée d'un système hétérogène à zones de mémoire partagées

On ne fait pas d'hypothèse sur le partage effectif de la mémoire au niveau matériel. Plusieurs logi-

ciels applicatifs peuvent évoluer dans le même espace mémoire physiquement partagé, sans exploiter cette caractéristique.

Mémoire partagée sous *VxWorks*

Le système d'exploitation *VxWorks* est destiné à l'embarqué et propose le support des plateformes multiprocesseurs comme une extension appelée *VxMP*. Cette extension permet de supporter les plateformes avec un maximum de 20 processeurs, éventuellement hétérogènes [23].

L'extension *VxMP* propose divers services supplémentaires, dont la mémoire partagée entre les processeurs. Le partage de mémoire n'est donc pas natif et demande l'appel à une *API* spécifique pour créer des partitions en mémoire partagée et allouer des zones de mémoire partagées. La synchronisation s'effectue à l'aide de *sémaphore*s spécifiques.

Mémoire partagée sous *RTEMS*

Le système d'exploitation temps réel *RTEMS*, destiné à l'embarqué, offre un support pour les plateformes multiprocesseurs hétérogènes également.

Il met en place une couche basse, proche du matériel, nommée *Shared Memory Support Driver*. Cette couche doit fournir les moyens d'établir des zones de mémoire partagées pour les couches plus hautes. Elle doit être développée spécifiquement pour chaque architecture supportée par le système d'exploitation et reste réservée à l'utilisation par les couches plus hautes du noyau.

2.1.3 Communication par messages

La communication entre les tâches par passage de messages, ou de paquets, est une technique courante dans les systèmes multiprocesseurs. Elle permet d'abstraire complètement les mécanismes matériels de communication à travers une *API* qui prend en charge les informations à faire transiter vers un autre sous-système de traitement de la plateforme (figure 2.4).

Cette technique, telle qu'on l'a expliquée précédemment, abstrait suffisamment le matériel pour fonctionner sur des systèmes hétérogènes ou même sur des *clusters* composés de différentes machines.

Messages sous *VxWorks*

VxWorks n'étant pas libre, les documents disponibles nous permettent de déduire que l'extension *VxMP* apporte déjà les services de partage de zones mémoire et propose également un service nommé *message queue* qui est utilisable avec les plateformes hétérogènes.

D'autres services concernant les systèmes multiprocesseurs et le multi-core sont en cours de développement dans le système d'exploitation, *Wind River* ayant pris conscience de l'essor de ces plateformes [23].

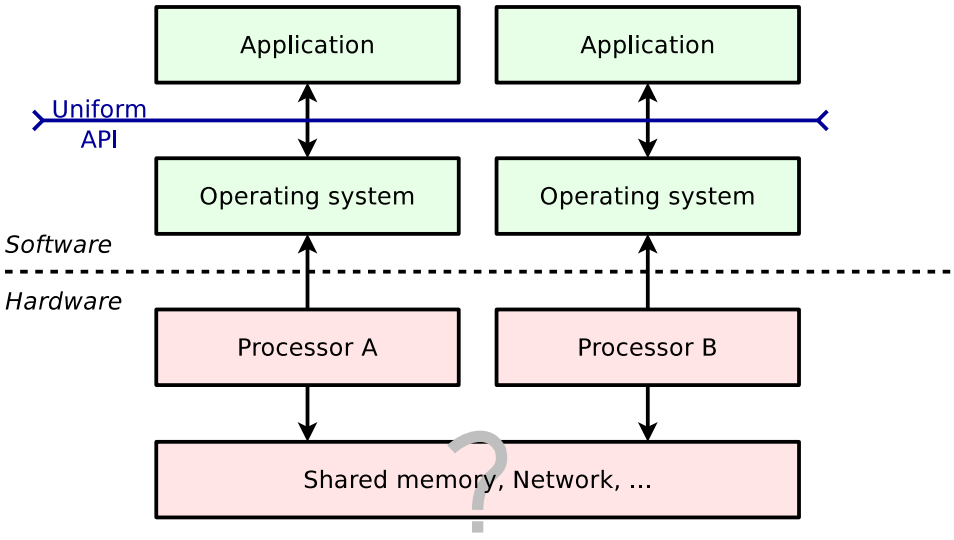

FIGURE 2.4 – Architecture simplifiée d'un système hétérogène à passage de messages

Messages sous *RTEMS*

Le passage de messages sous *RTEMS* est un service de haut niveau pris en charge par la couche *MPCI* (MultiProcessor Communications Interface) [5] qui s'appuie sur la couche *Shared Memory Support Driver* déjà évoquée.

MPCI propose une interface simple composée de fonctions telles `user_mpci_send_packet()` ou `user_mpci_receive_packet()` qui prennent comme argument un pointeur vers le paquet et un identifiant de noeud pour l'envoi. Chaque paquet commence par un en-tête obligatoire indiquant la taille du paquet et le nombre de mots de 32 bits qui doivent subir une conversion d'endianness, en fonction des processeurs en présence.

Les différents problèmes liés à l'hétérogénéité de l'endianness et de l'alignement des données, imposent les contraintes suivantes :

– L'adresse des paquets doit être alignée.
– La conversion d'endianness est prise en charge par *MPCI* si les paquets sont constitués d'entiers de 32 bits contigus.

Messages sous *DNA/OS*

Le système d'exploitation *DNA/OS* développé au laboratoire *TIMA* dispose d'une architecture modulaire à base de composants et supporte les systèmes sur puce multiprocesseurs hétérogènes[17]. La communication entre les tâches sur une plateforme hétérogène est basée sur le passage de messages et est prise en charge par les pilotes.

Messages sous *CoMOS*

Le système d'exploitation *CoMOS*[18] pour plateformes hétérogènes est basé sur une approche à composants communiquant par messages. Ce système se destine aux applications embarquées, orien-

tées capteurs et communication, mettant en œuvre des petits processeurs capables de réveiller des processeurs plus performants quand arrive le besoin de traiter des données.

Il propose une abstraction logicielle qui permet de décrire une application comme un ensemble de tâches asynchrones qui interagissent exclusivement par l'échange de messages en réponse aux événements extérieurs. Les tâches sont interconnectées via leur port d'entrée et leur port de sortie.

La bibliothèque MPI

MPI est certainement la bibliothèque de passage de messages la plus répandue. Elle est portée sur un très grand nombre de systèmes.

Cette bibliothèque permet d'échanger des paquets de données constituées de tous types. Elle abstrait les différents processeurs de la plateforme hétérogène à mémoire partagée ou répartie, en noeuds de calcul et prend en charge la conversion des types du langage. Les types complexes définis par l'utilisateur, comme les structures en C, sont également pris en charge : des appels permettent d'informer *MPI* des détails des structures de données pour assurer la conversion lors des échanges de messages, malgré les contraintes d'alignement et d'endianness.

2.1.4 Machines virtuelles hétérogènes

L'utilisation d'un langage qui ne s'exécute pas nativement sur les processeurs de la plateforme permet une autre forme d'abstraction capable de surmonter les problèmes liés à l'hétérogénéité (figure 2.5).

Le langage *Java*, par exemple, est compilé en un *bytecode* qui est ensuite exécuté par un logiciel de machine virtuelle. Ce langage se veut extrêmement portable puisqu'il suffit d'implémenter le logiciel de machine virtuelle sur une architecture de processeur pour être capable d'exécuter n'importe quel programme *Java* déjà compilé. Cette portabilité est un premier point en faveur du support de l'hétérogénéité.

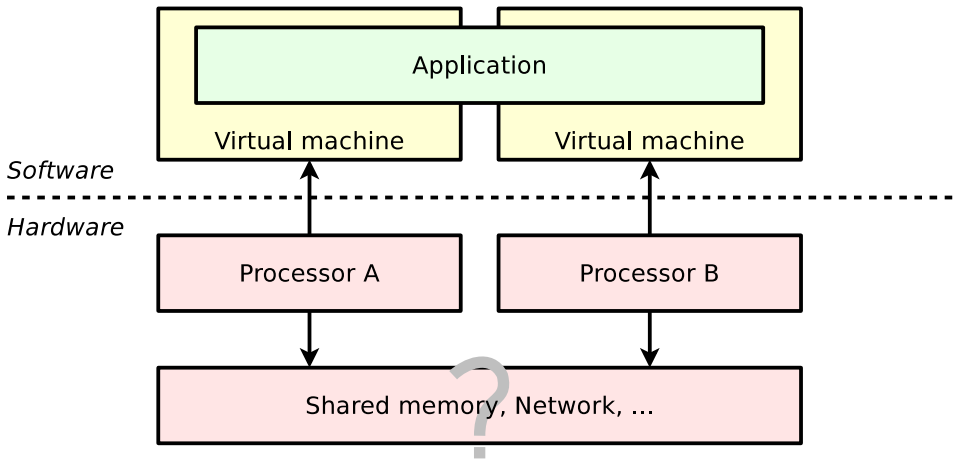

FIGURE 2.5 – Architecture simplifiée d'un système hétérogène à base de machine virtuelle

Les *opcode*s qui composent le *bytecode Java* effectuent tantôt des opérations similaires aux opérations implémentées dans les processeurs matériels, tantôt des opérations système liées à la gestion de la mémoire et des objets *Java*. Plusieurs développements existent pour implémenter une machine virtuelle java au-dessus d'un système parallèle ou distribué [10]. Parmi ces implémentations, certains projets s'intéressent aux systèmes multiprocesseurs hétérogènes.

CellVM

Le projet *CellVM* [26] est une adaptation du projet *JamVM* pour la puce Cell [4] d'*IBM* qui embarque des processeurs différents.

Deux types de machines virtuelles sont développés spécifiquement pour les deux types de processeurs : pour le coeur *PowerPC* principal et pour les 8 unités *SPE* du *CEll*. Ces machines virtuelles se nomment *ShellVM* et *CoreVM*. Toutes les instructions du *bytecode Java* ne peuvent pas être exécutées par *CoreVM* et doivent être prises en charge par le processeur principal, c'est le cas des opérations de gestion de la mémoire.

Le placement des tâches *Java* est définitif car la machine virtuelle répartie ne supporte pas la migration.

Hera-JVM

Hera-JVM [24] est un projet plus récent basé sur *JikesRVM* qui apporte certaines améliorations par rapport à *CellVM* :

– La machine virtuelle qui s'exécute sur les unités *SPE*, n'est pas spécifique et peut prendre en charge l'exécution de toutes les instructions du bytecode *Java*.
– La migration d'une tâche entre les processeurs est prise en charge.

2.1.5 Langages dédiés à la parallélisation

Différentes approches proposent un langage permettant d'exprimer le parallélisme d'algorithmes directement (figure 2.6), et laissent le soin aux outils de compilation de répartir la charge de calcul sur les différents coeurs disponibles dans la plateforme hétérogène. Parmi ces différentes approches, certaines proposent la parallélisation d'une petite partie seulement du code de l'application, d'autres tentent au contraire d'en paralléliser l'ensemble.

Parallélisation de noyaux de calcul

Les plateformes hétérogènes orientées parallélisation de calculs, constituées par les processeurs centraux et les processeurs graphiques des PC récents, sont exploitées par différents environnements logiciels basés sur un principe identique :

– *CUDA*[3] est l'environnement de développement propriétaire pour le matériel *Nvidia*.
– *Close to Metal*[9] est l'environnement de développement propriétaire pour le matériel *AMD/Ati*.

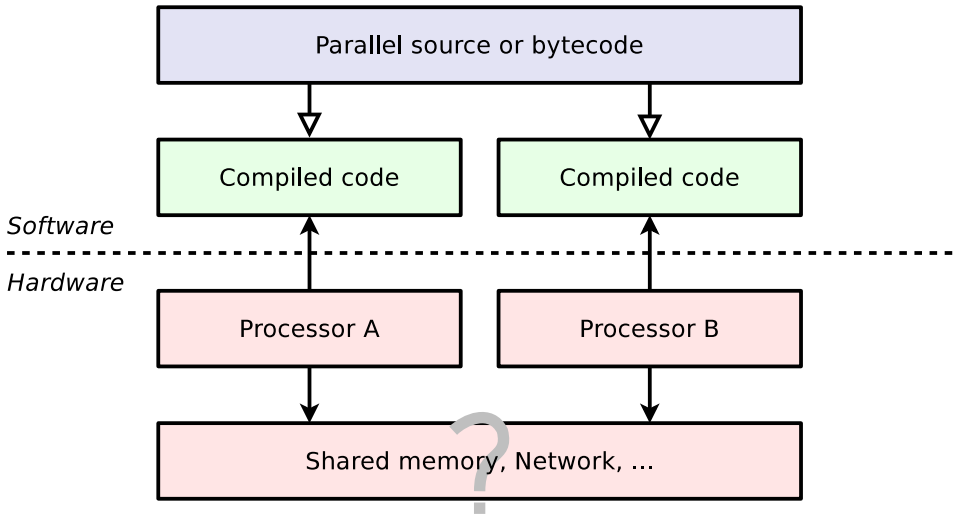

FIGURE 2.6 – Architecture simplifiée d'un système avec hétérogénéité prise en charge au niveau langage

– *OpenCL* est un standard émergeant qui vise à remplacer les deux environnements propriétaires cités.

Tous ces environnements permettent la parallélisation des calculs au niveau instructions et au niveau tâches. Ils proposent un sous-ensemble du langage C qui doit être utilisé pour implémenter un noyau de calcul, lequel sera exécuté de manière parallèle, selon les ressources de la machine.

– Le noyau de calcul est compilé à la volée pour les divers processeurs cibles de la plateforme, en essayant de tirer parti du parallélisme d'instructions.
– Le code compilé spécifiquement est exécuté de manière parallèle dans plusieurs tâches logicielles sur les processeurs disponibles.

OpenCL utilise un langage C restreint pour le noyau de calcul :
– pas de pointeur sur fonction.
– pas de pointeur passé en arguments.
– pas de récursivité.
– pas de type de données complexes.
– limitation des types entiers et flottants disponibles
Ces environnements sont donc adaptés à la parallélisation d'un unique algorithme de traitement de données réparti et ne permettent à priori pas la mise en œuvre d'une autre forme de parallélisme, telle qu'un graphe de tâches communicantes.

Le but d'*OpenCL* est de faciliter le développement d'applications exploitant le parallélisme des PC modernes, notamment pour les développeurs non initiés aux techniques avancées de programmation répartie qui impliquent la synchronisation explicite des tâches.

Le framework *Merge*

Merge[21] est un modèle de programmation destiné aux systèmes multi-core hétérogènes. Il prend en charge la sélection de l'implémentation d'algorithmes de calcul la plus appropriée à l'architecture,

et propose un langage de programmation qui permet d'exprimer le parallélisme.

Le langage mis en place par *Merge* est basé sur une approche dite *Map/Reduce* qui subdivise récursivement l'algorithme de traitement en sous-problèmes pour ensuite les affecter aux différents processeurs de la plateforme.

Merge implémente le compilateur du langage ainsi que le logiciel rendant possible l'exécution de l'application répartie sur la plateforme.

2.1.6 Conclusion sur les solutions hétérogènes

Les différentes approches que l'on vient de présenter traitent le problème de l'hétérogénéité en intervenant à différents niveaux de la chaîne de développement du logiciel :

– A travers des bibliothèques et interfaces logicielles imposées qui permettent de répartir l'application.
– Au niveau du langage de programmation qui permet la parallélisation des applications sur plateformes hétérogènes.

Cette prise en charge de l'hétérogénéité qui n'est pas native pose les problèmes suivants :

– Les approches qui proposent des mécanismes de communication spécifiques entre les tâches s'exécutant sur des processeurs de différents types, figent le déploiement du logiciel très tôt dans le cycle de conception. L'architecture du logiciel et le graphe des tâches de l'application dépendent du matériel, ce qui aboutit à l'écriture de code spécifique. Ceci rend difficile l'exploration architecturale et casse la flexibilité de redéploiement ou de déploiement tardif des tâches sur les processeurs.
– Ces solutions imposent plusieurs contraintes aux développeurs d'applications en exigeant une *API* ou un langage spécifique, lesquels limitent ou empêchent la réutilisation de code existant initialement développé pour des plateformes multiprocesseurs homogènes classiques. Le code existant étant généralement basé sur des langages et bibliothèques largement répandues comme le *C* et les *thread*s *POSIX*.

La solution qui est présentée dans ce rapport se base sur un noyau de système d'exploitation conçu pour résoudre le problème de l'hétérogénéité de manière native. La conception flexible de ce noyau et les choix d'architecture logicielle sont déterminants dans notre cas.

2.2 Les différents types de noyaux d'OS

Parmi les différentes familles d'architectures de noyaux de systèmes d'exploitation, certaines possèdent des caractéristiques plus ou moins bien adaptées à la gestion de l'hétérogénéité des processeurs.

2.2.1 Séparation et modularité

L'architecture d'un noyau de système d'exploitation intervient à plusieurs niveaux de la conception du logiciel. Selon les besoins certains points peuvent être traités de différentes manières, particulièrement soignés ou laissés de côté :

- Le niveau de sécurité et de robustesse du cloisonnement des services.
- La finesse de partage et de multiplexage des ressources du système.
- La généricité permettant la portabilité du noyau sur plusieurs architectures.
- D'autres contraintes spécifiques, dans le cas présent, la prise en compte des contraintes d'hétérogénéité des processeurs.

Le souci de sécurité implique une séparation des privilèges et un cloisonnement plus ou moins important des applications mais aussi des services système, comme la gestion des fichiers ou encore la pile réseau. Cette séparation s'appuie sur les différents niveaux de privilèges d'exécution supportés matériellement par le processeur.

Le partage et le multiplexage des ressources du système impliquent, quant à eux, un certain effort de recherche dans le choix et la mise en œuvre d'algorithmes efficaces d'allocation et d'ordonnancement.

La généricité et la portabilité du code passent par une certaine modélisation logicielle avec des couches et des *API* bien définies, et donc, une certaine organisation et modularité du code source. Cette modularité peut éventuellement se retrouver au niveau binaire lorsque le noyau supporte le chargement dynamique de modules ; à l'inverse, elle peut disparaître à la compilation. Les grandes classes d'architectures de noyaux sont définies par certains de ces critères seulement, si bien qu'il peut facilement exister des approches différentes dans une même classe ou encore des conceptions hybrides.

La suite présente les grandes classes d'architectures que sont les noyaux monolithiques et les micro-noyaux, ainsi que les solutions moins répandues telles que les exo-noyaux.

2.2.2 Approche monolithique

La conception monolithique d'un noyau de système d'exploitation est la plus simple et la plus répandue aujourd'hui (figure 2.7). C'est elle qui présente le moins de modularité.

Dans les noyaux monolithiques qui proposent une séparation des privilèges, seuls deux domaines d'exécution existent : Le noyau qui possède tous les privilèges et l'application qui s'exécute avec moins de privilèges. Tous les services noyau s'exécutant avec les mêmes privilèges, un bogue dans un pilote de périphériques, par exemple, peut mettre en péril le fonctionnement de tout le système.

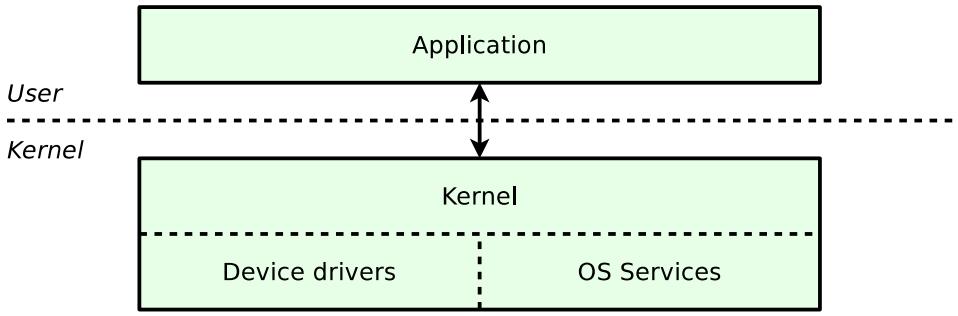

FIGURE 2.7 – Architecture d'un noyau monolithique

L'organisation du code varie d'un noyau à l'autre ; dans certains projets on rencontre une couche d'abstraction du matériel permettant la portabilité sur différentes architectures matérielles.

Les noyaux suivants, comme beaucoup d'autres, sont classables dans cette catégorie : *Linux*, *NetBSD*, *OpenBSD*, *FreeBSD*...

2.2.3 Les micronoyaux

Dans l'architecture à base de micronoyau, le noyau est très réduit et prend seulement en charge le multiplexage et l'allocation des ressources de base que sont le temps processeur et la mémoire. Tous les autres services, comme la gestion des fichiers, la pile réseau ou les pilotes de périphériques, sont cloisonnés dans des niveaux d'exécution ayant des niveaux de privilèges inférieurs (figure 2.8).

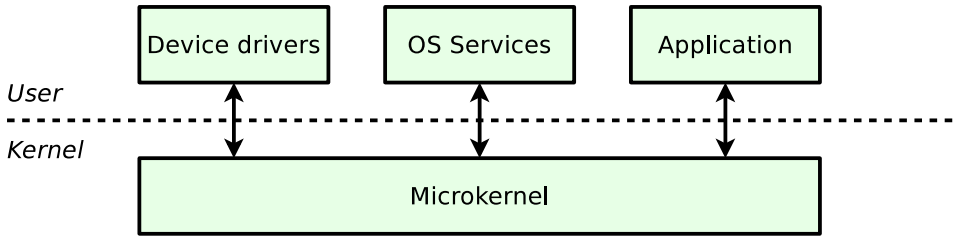

FIGURE 2.8 – Architecture d'un micronoyau

Cette approche présente un intérêt lorsque la sécurité est primordiale, mais les performances sont souvent inférieures à l'approche monolithique en raison des canaux de communication moins directs entre les services

Cette approche n'a pas de sens dans les cas où la séparation des privilèges n'est pas mise en œuvre

Les micronoyaux *Mach* ou les différentes variantes de *L4* sont des exemples caractéristiques de ce type de système.

2.2.4 Les noyaux hybrides

On parle souvent de noyau hybride lorsque l'on est en présence d'une architecture intermédiaire entre les noyaux monolithiques et les micronoyaux. Dans ce cas, une partie des services qui pourrait

être cloisonnée s'exécutent en mode noyau pour des raisons de performance. C'est le cas des noyaux de la série des systèmes *Windows*. Il peut cependant exister bien d'autres formes d'hybridation des noyaux de systèmes d'exploitation, tant les possibilités de variations sur les choix de conception sont grands.

2.2.5 Les exo-noyaux

L'architecture en exo-noyau s'attache à abstraire et multiplexer les différentes ressources du système pour les présenter à différentes bibliothèques qui implémentent des interfaces utilisateur prédéfinies et/ou standardisées. Un système à base d'exo-noyau sépare donc nettement, au moins dans sa modélisation logicielle, la gestion des ressources et les interfaces qui permettent de les proposer aux applications (figure 2.9).

On peut, par exemple, imaginer qu'un système à base d'exo-noyau propose à la fois une interface *UNIX* et une interface *L4*, simultanément utilisables par des applications adaptées à chaque type de système.

L'architecture en exo-noyau n'est donc pas définie par la séparation des privilèges qu'elle met en place à l'exécution, mais plutôt par l'organisation des couches logicielles qui permettent l'abstraction et la modularité nécessaires aux différentes facettes que peut présenter le noyau aux applications. L'architecture en exo-noyau caractérise plusieurs projets de recherche notamment *ExOS* et *Xok* étudiés au *MIT* [14] [15].

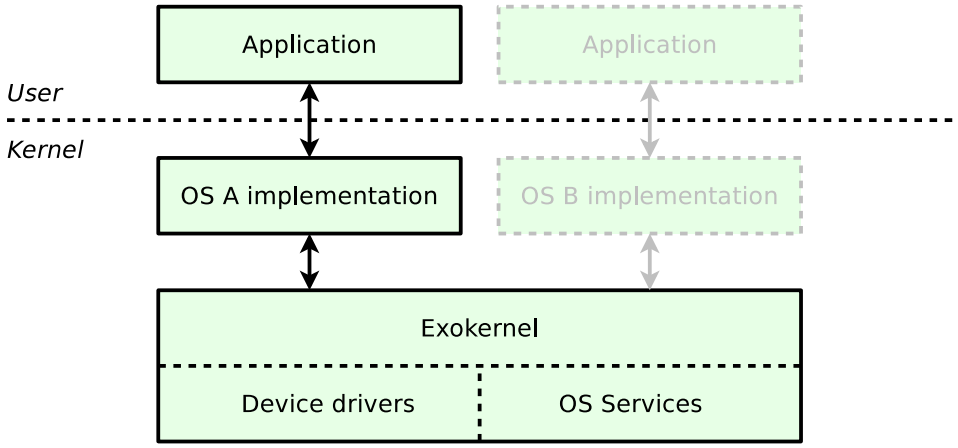

FIGURE 2.9 – Architecture d'un exo-noyau

La modularité évoquée ici est liée à l'organisation et à la séparation des composants logiciels dans le code source. Une fois configuré et compilé, un tel noyau peut éventuellement revêtir l'aspect d'un noyau monolithique ou d'un micronoyau.

2.2.6 Conclusion sur les architectures noyaux

Les problèmes liés à l'hétérogénéité interviennent principalement au niveau du matériel. L'aspect le plus important de l'architecture du noyau d'un système supportant nativement les plateformes

hétérogènes semble donc lié à la portabilité, au multiplexage des ressources et à l'abstraction du matériel.

La question de la séparation des privilèges apparaît comme secondaire, car sans conflit avec le support de l'hétérogénéité.

L'architecture en exo-noyau est l'approche retenue pour supporter nativement l'hétérogénéité des processeurs. En effet, cette architecture logicielle modulaire permet la plus forte abstraction. Cette modularité du logiciel est adaptée à la gestion et au multiplexage des ressources matérielles par les couches basses du noyau, dont les processeurs, tout en rendant l'hétérogénéité abstraite pour les couches plus hautes.

Cette architecture doit donc permettre de confiner le support de l'hétérogénéité dans l'exo-noyau, tandis que plusieurs implémentations de systèmes d'exploitation, interfacées avec les applications adaptées, peuvent profiter de l'hétérogénéité de manière transparente.

Chapitre 3

Solutions de principe

Cette partie présente les solutions techniques retenues pour l'élaboration du noyau d'un système d'exploitation capable de gérer nativement l'hétérogénéité sur une plateforme matérielle multiprocesseur hétérogène à mémoire partagée.

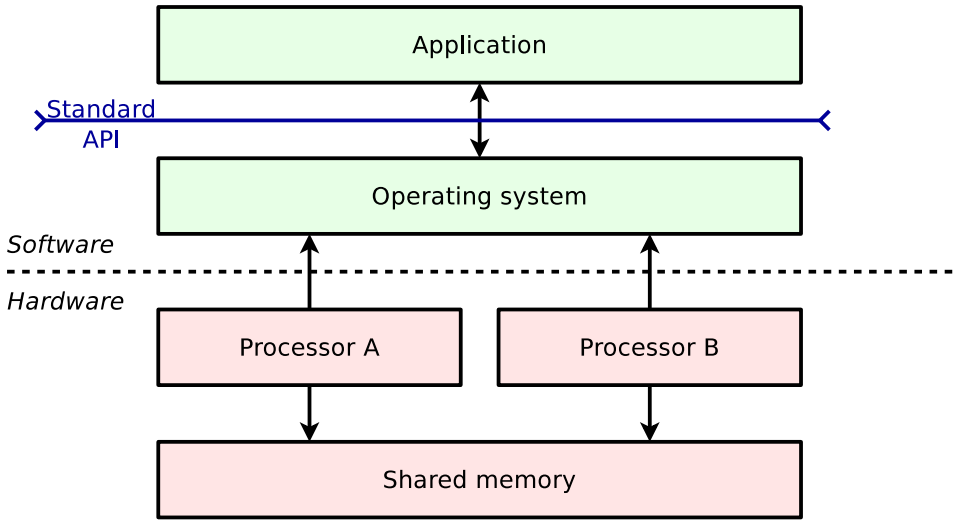

FIGURE 3.1 – Architecture simplifiée de plateforme avec prise en charge native de l'hétérogénéité

La figure 3.1 schématise l'architecture de l'ensemble du système d'une manière comparable aux figures des autres architectures présentées dans le chapitre 2.

3.1 Architecture du noyau

L'architecture du noyau doit mettre en place une couche d'abstraction permettant de masquer les différences matérielles. Comme dans les noyaux portables classiques, le choix entre plusieurs plateformes supportées peut s'effectuer à la compilation. Cependant la prise en charge de plusieurs architectures de processeurs dans une même plateforme impose une capacité d'abstraction à l'exécution.

L'abstraction et le multiplexage de ressources matérielles, comme les périphériques divers, constituent des services classiques rendus par les noyaux de systèmes d'exploitation ; il n'est pas rare de voir cohabiter différents modèles d'un même type de périphérique, dans une plateforme matérielle. Notre approche doit permettre de résoudre le même problème avec les processeurs.

Du point de vue logiciel, la ressource fournie par les processeurs est le temps de calcul. Ce temps de calcul est généralement multiplexé ou réparti entre les tâches par le noyau du système d'exploitation.

Il existe plusieurs bibliothèques de haut niveau, utilisables directement par les applications, permettant de découper le travail en tâches logicielles : la bibliothèque des *thread*s *POSIX* ou *OpenMP* par exemple. Celles-ci doivent pouvoir être implémentées sans interférer avec l'hétérogénéité des processeurs que l'on veut rendre transparente. La prise en charge de la plus grande diversité d'applications déjà existantes est un des objectifs du support natif de l'hétérogénéité.

L'architecture en exo-noyau [14] est retenue pour le noyau supportant nativement l'hétérogénéité qui est proposée ici. C'est en effet ce choix qui apporte le plus de flexibilité en termes d'interface avec l'applicatif, tout en permettant la mise en place d'une couche d'abstraction des processeurs et de la plateforme.

Cette architecture en exo-noyau n'est pas encore répandue dans les systèmes d'exploitation que l'on rencontre en production, mais elle est étudiée et mise en avant dans divers travaux de recherche, en raison de la grande flexibilité qu'elle apporte.

Le noyau réalisé au cours de cette thèse dispose donc des caractéristiques suivantes, grâce à sa couche d'abstraction du matériel et à son architecture en exo-noyau :

- Gestion de l'hétérogénéité des processeurs de manière native.
- Portabilité sur différents processeurs et différentes plateformes.
- Possibilité de proposer des services standards, compatibles avec ceux d'autres noyaux, au niveau applicatif (*thread*s *POSIX* en mode noyau, *OpenMP*, *UNIX*, ...).

La couche d'abstraction du matériel se nomme *Hexo*, alors que l'ensemble du noyau avec ses diverses bibliothèques de services, de pilotes de périphériques et d'interfaces avec l'application, se nomme *MutekH*. La figure 3.2 présente l'architecture de *MutekH* déployé sur une plateforme comportant deux types de processeurs distincts.

Il faut préciser que l'application peut s'exécuter ou non en mode privilégié, car celles embarquées n'ont pas toujours besoin de cette protection. Lorsque l'application utilise une interface lui permettant de s'exécuter en mode noyau, elle peut accéder à toutes les ressources et utiliser les nombreuses bibliothèques de *MutekH* directement. Une application, qui s'exécute en mode utilisateur, n'a accès qu'aux ressources système mises à disposition par les appels système de son interface avec le noyau.

3.2 Les contraintes réalistes

La prise en charge de l'hétérogénéité matérielle de la plateforme, sur laquelle doit s'exécuter le code, peut être plus ou moins complexe en fonction des différences entre processeurs en présence. Certaines plateformes matérielles sont viables, d'autres ne le sont pas, d'autres encore impliquent un coût en performances ou une complexité accrue à cause des contraintes imposées par l'hétérogénéité.

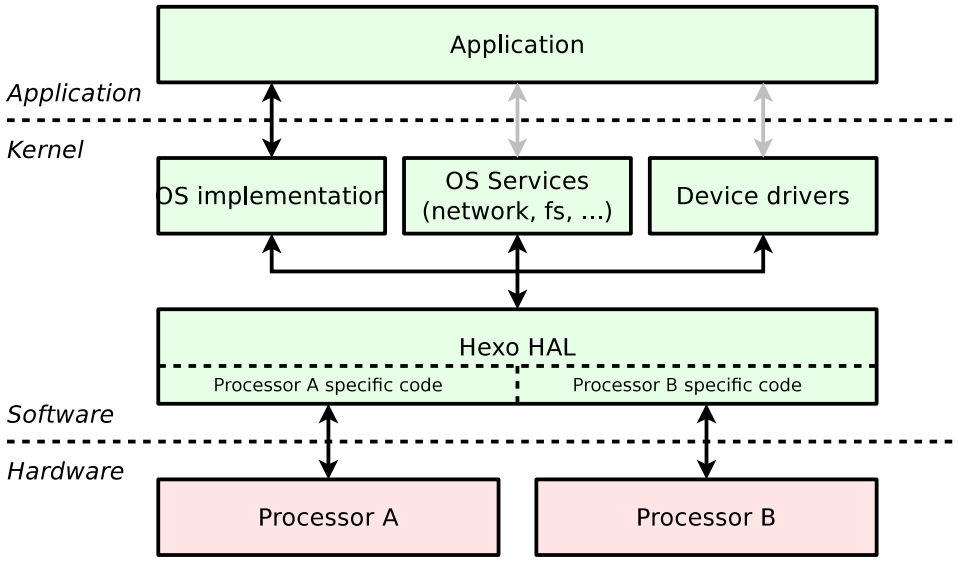

FIGURE 3.2 – Architecture de MutekH

Lorsque l'on se pose la question de l'intérêt d'un système hétérogène à mémoire partagée, on admet facilement que beaucoup de configurations matérielles, que l'on peut imaginer, ne répondent à aucun besoin réel. Considérons ainsi :

– Les plateformes matérielles hétérogènes existantes, qui sont déjà exploitées par l'une des solutions logicielles présentées dans le chapitre 2, retiennent évidemment notre attention et pourraient tirer un réel bénéfice d'un support natif de la part du système d'exploitation. Il s'agit généralement de plateformes employant des processeurs généralistes aux jeux d'instructions identiques et, éventuellement, de processeurs spécialisés tels que des *DSP*.

– Les plateformes matérielles hétérogènes où les processeurs sont volontairement choisis pour leurs différences ont un intérêt industriel limité, mais elles offrent un bon terrain d'expérimentation pour la mise au point et la recherche dans le domaine. C'est le cas des plateformes employant des processeurs généralistes aux jeux d'instructions différents.

– Les plateformes matérielles hétérogènes les plus exotiques disposent de processeurs différant sur des aspects qui limitent leur interopérabilité, impactant ainsi les performances de l'ensemble. C'est le cas par exemple d'une plateforme hétérogène employant des processeurs avec une endianness différente.

3.2.1 Le problème de l'endianness

La vision de la mémoire par un processeur dépend, entre autres, de l'ordre d'écriture des octets à l'intérieur d'un mot mémoire. Cet ordre est souvent déterminé intrinsèquement par le type de processeur qui réalise l'accès à la mémoire. C'est ainsi que l'on distingue les processeurs *big endian* et les processeurs *little endian*.

Sur certains processeurs, il est possible de paramétrer cet ordre grâce à un registre de configura-

tion. Cette fonctionnalité n'est cependant pas exploitée par les outils de génération de code pour un changement dynamique d'endianness, de plus elle n'existe pas sur tous les processeurs.

On doit donc évoquer le problème de la cohabitation de processeurs aux endianness différentes dans une plateforme hétérogène.

Ce problème s'apparente à celui de l'écriture d'informations binaires dans un fichier ou un paquet réseau, puisque l'on est également dans le cas où un processeur doit pouvoir lire les mots écrits par un processeur différent.

La solution communément adoptée pour résoudre ce problème courant, consiste à établir une convention sur l'ordre d'écriture des octets en mémoire. Si la convention impose, par exemple, l'accès des mots dans l'ordre *big endian* pour tel format de fichier, les processeurs nativement *little endian* devront permuter la valeur stockée, à l'aide d'opérations logiques, lors de l'accès mémoire.

Si cette approche est viable dans les cas très peu fréquents des accès disques et réseau, il est impensable d'avoir à modifier, dans le code source du système d'exploitation et de l'application, chaque accès mémoire. Une telle approche aurait les conséquences suivantes :

– Diminution des performances.
– Augmentation de la taille du code.
– Impact important sur la lisibilité du code source.

Si l'on accepte l'impact au niveau des performances et de la taille du code binaire, on peut envisager une modification du compilateur qui insérerait des permutations à **tous** les accès mémoire sur des mots. On obtiendrait ainsi un processeur qui se comporterait avec une endianness non native. Il apparaît donc que la cohabitation de processeurs aux endianness différentes n'est pas incontournable mais aurait un impact sur les performances et la complexité de la mise en œuvre. Cet état de fait concernant le coût des conversions d'endianness n'est pas spécifique au support natif de l'hétérogénéité dont il est question ici ; il est généralement rencontré lors de l'accès aux périphériques, par les pilotes.

On fait le choix de ne pas traiter le problème de l'endianness dans la solution proposée, en raison des contraintes évoquées qui limitent l'intérêt d'une telle plateforme.

3.2.2 Largeur de mot

La largeur de mot exploitée par un processeur est généralement déterminée par la taille de ses registres généraux, elle est de 8, 16, 32 ou 64 bits.

Les systèmes d'exploitation courants, et les logiciels d'une manière plus générale, sont développés pour une largeur de mot donnée et ne fonctionnent pas sur des processeurs ayant une largeur de mot différente. Au mieux lorsque le code source est développé avec soin, il est possible de le compiler et de le faire fonctionner sur un processeur ayant une largeur de mot supérieure, comme on le voit souvent avec les évolutions de systèmes 32 vers 64 bits.

Lors du développement d'un noyau, plusieurs questions se posent relativement à la largeur de mot :

– Quelles largeurs de mots doivent être supportées par l'implémentation du noyau ?
– Doit-on supporter plusieurs largeurs de mots différentes dans une même plateforme hétérogène ?

– Quelles sont les contraintes soulevées et le coût en performances des différentes options ?

Un usage judicieux des types entiers du langage C permet d'aborder le problème.

Les types du langage

Le degré de portabilité du code que l'on rencontre généralement n'est pas extrême, aujourd'hui les programmes sont fréquemment conçus pour les processeurs ayant une largeur de mot de 32 bits, souvent portables vers 64 bits. Il est cependant possible de supporter une plus large gamme de largeurs d'entiers natifs dans le code source d'un logiciel, avec un peu de rigueur et une utilisation judicieuse des types de données du langage.

Les types entiers du langage C, utilisés en pratique, déterminent la largeur des mots que l'on manipule. Ces types n'ont pas une largeur fixe d'une architecture de processeur cible à une autre. Un des principaux problèmes de portabilité rencontré concerne la conversion (*cast*) entre les types entiers et les pointeurs. Ainsi, si on développe pour un processeur 32 bits, le type `int` possède la même taille qu'un pointeur bien que ce ne soit plus le cas sur un système 64 bits, par exemple.

Il convient donc de ne pas utiliser les types entiers natifs du compilateur (`int`, `long`, ...) et d'utiliser des types dont la largeur reste fixe quelle que soit l'architecture du processeur cible.

Si l'on cherche également la portabilité vers les systèmes ayant des largeurs de mots inférieures, le problème se complique encore. Un autre comportement courant chez les développeurs est d'utiliser le type entier `int` pour chaque variable entière d'un algorithme, et ce, sans se poser la question du dimensionnement au plus juste des types.

En effet, lorsque l'on est en présence d'une variable entière qui ne prend pas de valeur plus grande que 100, il n'est pas nécessaire d'employer le type `int` ou tout autre type d'une largeur de 32 bits. C'est pourtant ce type qui permet la génération du code le plus efficace sur une machine 32 bits. Cependant sur un processeur 8 bits, l'utilisation du type `int` impose au compilateur l'allocation d'au moins deux registres et l'astreint au déroulement de plusieurs instructions avec propagation de retenues pour chaque opération sur cette variable plus large qu'un registre ; or un seul registre aurait suffit pour traiter la valeur maximale utilisée dans l'algorithme.

Une fois encore, il apparaît que les types natifs du compilateur C ne sont pas adaptés à la portabilité extrême du code source. Il faut donc utiliser des types entiers nommés d'après la largeur minimale que l'on souhaite employer, mais qui peuvent être surdimensionnés sur certains processeurs pour des raisons de performance.

Le standard *Iso C99* propose tous les types nécessaires pour résoudre ces problèmes. Ceci permet donc d'écrire du code portable sur les architectures de 8 et 64 bits sans compromis sur les performances, moyennant un effort minimum de rigueur dans le développement.

Outre les problèmes classiques de portabilité et de performances, l'hétérogénéité apporte une contrainte supplémentaire sur la largeur des types entiers.

Largeur de mot hétérogène

Dans un système hétérogène à mémoire partagée, il faut prendre un soin particulier à harmoniser les structures de données en mémoire. En effet, le code source compilé pour différentes architectures de processeur doit être configuré pour que les types entiers employés dans l'ensemble du noyau et de l'application, possèdent la même largeur. Ainsi les structures et les sections de données en mémoire possèdent la même empreinte.

Dans le cas d'un système hétérogène où deux processeurs n'ont pas la même largeur de registres, les types entiers pour le processeur disposant de la plus petite taille de registres doivent être surdimensionnés.

Cependant, ce surdimensionnement implique un coût supplémentaire en performances par la charge accrue qu'il impose : à la compilation, sur l'allocation des registres et à l'exécution, sur la propagation des retenues dans les opérations arithmétiques courantes.

En dehors de ce premier problème qui limite l'intérêt d'une telle plateforme, sans toutefois entraver sa réalisation technique, se pose un second problème concernant les accès atomiques à la mémoire.

L'accès à une valeur par le processeur possédant la plus grande largeur de mot peut nécessiter plusieurs accès pour le second processeur doté d'une largeur de mot plus petite. La multiplicité de ces accès peut devenir problématique lorsqu'il s'agit de réaliser un unique accès atomique à la mémoire.

Ce problème n'est pas incontournable puisqu'on peut envisager de n'utiliser que la partie basse du mot pour effectuer les opérations d'accès atomique, ce qui est suffisant pour réaliser les opérations courantes sur les verrous.

La gestion de ce genre de plateforme ne pose donc pas de défi technique supplémentaire par rapport aux plateformes à largeur de mot homogène. La conception du noyau dont il est question ici n'écarte pas cette éventualité, cependant aucune expérimentation n'a été entreprise dans ce sens.

Conclusion sur la largeur des types

La solution retenue est une utilisation **exclusive** des types standards *ISO C99* pour les entiers. Ces types ont été standardisés pour résoudre précisément les problèmes de portabilité et permettent également d'aborder l'hétérogénéité :

- Ils ne sont pas natifs et leur définition dans le code source du noyau peut être différente selon les processeurs en présence.
- Leur définition selon la configuration hétérogène permet de garantir que la largeur des mots est uniforme entre les processeurs tout en restant optimale.

3.2.3 Différences de jeux d'instructions

La différence des jeux d'instructions des différents processeurs est la première contrainte évidente lorsque l'on s'intéresse aux logiciels devant supporter l'hétérogénéité matérielle des processeurs.

Entre deux processeurs différents, le format binaire de chaque instruction est complètement différent. De plus, certains processeurs disposent d'instructions qui ne sont pas disponibles sous la même forme ou avec le même mode d'adressage que sur le processeur voisin.

Faire fonctionner un logiciel réparti sur les différents processeurs, nécessite de compiler le logiciel plusieurs fois, pour chaque architecture de processeur présente dans la plateforme. Il faut toutefois prendre toutes les précautions pour s'assurer que le logiciel impose strictement le même comportement à tous les processeurs, sans quoi la cohabitation n'est pas possible.

Lorsqu'on dispose des multiples codes binaires compilés du même code source, l'encodage des instructions ou la disponibilité de telle ou telle instruction d'un processeur à l'autre n'est plus un problème. Le vrai problème à résoudre concerne la localisation des symboles en mémoire, code et données.

Si au niveau fin on accepte que les instructions soient différentes, ce n'est pas le cas au niveau des fonctions. Dans un logiciel, la fonction est l'entité de code la plus petite qui est visible globalement. Pour une fonction source identique, ses instances compilées en mémoire doivent non seulement se comporter de manière identique d'un processeur à l'autre, mais aussi se situer aux mêmes adresses pour tous les processeurs. Cette contrainte d'adresse est indispensable pour qu'un pointeur sur fonction, qui est une donnée partagée entre processeurs, désigne la même fonction quel que soit le type de processeur.

Il est intéressant de se demander si la différence des jeux d'instructions entre processeurs en présence dans une plateforme hétérogène est toujours nécessaire, bien qu'envisageable. Il est certain qu'on ne peut pas prétendre supporter nativement les plateformes multiprocesseurs hétérogènes sans apporter une solution élégante à ce problème. Mais cette solution ne permettrait-elle pas de supporter également des codes binaires compilés différemment pour des processeurs aux jeux d'instructions identiques ?

Si on liste les plateformes qu'on peut rencontrer dans de vraies applications, on peut dégager les cas suivants :

– Les plateformes disposant de processeurs généralistes du même type et de processeurs spécifiques comme les DSP.
– Les plateformes disposant de un ou quelques processeurs superscalaires haute performance et de beaucoup de processeurs plus petits, tous dotés du même jeu d'instruction de base.

Lorsque tous les processeurs possèdent le même jeu d'instruction de base, il n'est pas toujours intéressant de leur faire exécuter la même version binaire du code source compilé :

– L'architecture interne des processeurs peut être différente, ce qui nécessite un ordonnancement différent des instructions par le compilateur pour obtenir les meilleures performances.
– Certains processeurs peuvent disposer d'extensions du jeu d'instructions (virgule flottante, *SIMD*, ...), impactant également la génération de code et l'optimisation.

3.2.4 Virtualisation de la mémoire de code

On sait donc qu'un des principaux défis du support natif de l'hétérogénéité concerne la vision identique de la mémoire, que doivent avoir les processeurs. Ainsi, chaque symbole (variable, fonction, etc.) doit se situer aux mêmes adresses pour chaque type de processeur.

Si on admet que les données peuvent et doivent être complètement partagées, le code des mêmes fonctions, compilé différemment pour les différents types de processeurs, ne peut l'être.

Pour pouvoir localiser les différentes instances d'une même fonction source à la même adresse en mémoire, on doit virtualiser la mémoire contenant le code.

De cette manière, lorsqu'un processeur accède à son code, il peut lire la version compilée qui lui est destinée et qui n'intéresse que lui. On peut alors imaginer l'agencement des binaires de chaque processeur pour que les mêmes fonctions soient aux mêmes adresses, et qu'un pointeur sur fonction soit valable pour tous les processeurs, condition nécessaire au partage des données.

La solution retenue se base donc sur deux techniques : la virtualisation de la mémoire de code et la génération de fichiers binaires aux adresses uniformisées entre les processeurs.

La plateforme matérielle intégrant les processeurs doit donc mettre en place un mécanisme de routage matériel capable de rediriger les accès mémoire dans le segment de code vers différents composants, et ce, en fonction du type de processeur qui émet la requête. La figure 3.3 illustre la correspondance mémoire d'une telle plateforme.

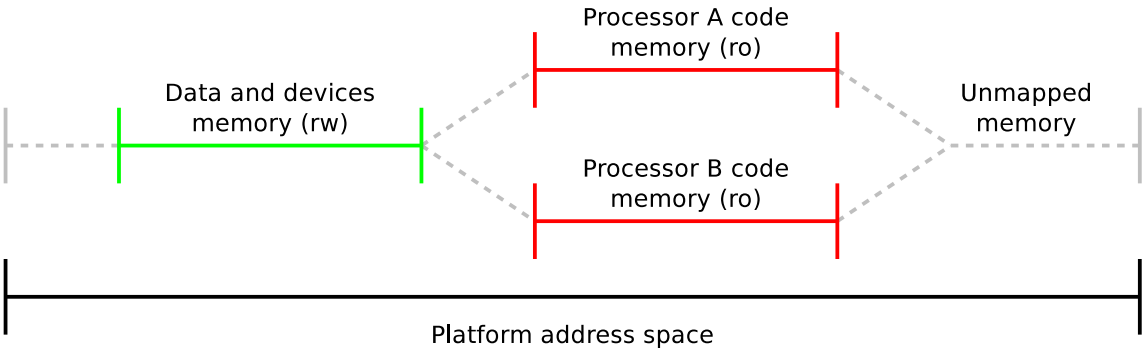

FIGURE 3.3 – Virtualisation du segment de code dans l'espace d'adressage de la plateforme

Il existe deux approches évidentes pour apporter le support de la virtualisation :

– Dans les systèmes sans mémoire virtuelle, le routage des requêtes vers des mémoires différentes peut se faire selon le processeur source, à l'aide d'un composant matériel simple réalisant le démultiplexage.
– Lorsque le processeur dispose d'une *MMU*, on utilise la mémoire virtuelle. Dans ce cas le système d'exploitation met en place le mapping mémoire adéquat lors de l'initialisation du système.

La mémoire de code étant en lecture seule, on considère que l'on est encore dans le cas d'une plateforme à mémoire partagée, car toute la mémoire accessible en écriture, permettant la communication, reste effectivement partagée entre les processeurs. Le développement d'applications portables se base sur cette hypothèse ; on peut donc exclure les techniques de code auto-modifiant.

3.3 Plates-formes matérielles cibles

Le choix d'une ou plusieurs plateformes matérielles cibles, supportées par un noyau détermine en partie son degré de portabilité. La conception du noyau dont il est question ici le destine à supporter divers types de plateformes :

– Des plateformes embarquées monoprocesseurs, à base de microcontrôleurs, par exemple.

– Des plateformes multiprocesseurs classiques et hétérogènes.
– Des plateformes massivement parallèles many-core hétérogènes.

On peut estimer que, si le noyau capable d'hétérogénéité est également capable de remplir la tâche d'un noyau classique avec le même niveau de performances, il pourra apporter le même niveau de services lorsqu'il s'exécutera sur une plateforme hétérogène. Le projet consiste donc bien à élaborer un système d'exploitation utilisable dans des conditions réelles d'applications parallèles qui s'exécutent sur des architectures multiprocesseurs, et non seulement un noyau capable de mener à bien les expérimentations propres au support natif de l'hétérogénéité.

Le choix de portage sur des plateformes matérielles bien différentes doit garantir la robustesse de l'abstraction du matériel mise en place, permettant ainsi de traiter le problème du développement du noyau avec toutes les contraintes réelles, sans se contenter d'un sous-ensemble matériel plus simple.

La couche d'abstraction du matériel dans le code du noyau va plus loin que celle que l'on rencontre habituellement dans les autres noyaux : le code spécifique à la plateforme est séparé du code spécifique au microprocesseur.

Si les plateformes multiprocesseurs homogènes classiques sont disponibles facilement (PC à base de multiprocesseurs *x86*), ce n'est pas le cas des plateformes multiprocesseurs hétérogènes à mémoire partagée.

Les plateformes hétérogènes utilisées pour les expérimentations sont mises en place grâce au projet *SoCLib*, bibliothèque de composants en *SystemC* permettant l'élaboration d'architectures multi-core et many-core proposant de nombreux types de coeurs de processeurs, de périphériques et de composants d'interconnexion. Cette bibliothèque permet de créer des simulateurs de toutes sortes dont les caractéristiques sont paramétrables finement, elle dispose de modules permettant le debug logiciel et matériel. Elle reste modifiable au besoin, comme tout logiciel libre.

SoCLib est une bibliothèque de simulation utilisée en microélectronique, la modélisation de la plupart des composants reflète leur comportement au niveau électronique. La simulation est donc beaucoup plus fine que celle proposée par les solutions d'émulation et de virtualisation telles que *QEmu*, qui favorisent les performances au détriment de la précision. Certains composants *SoCLib* sont également disponibles en description matérielle *Vhdl* pour l'exécution sur puce *FPGA*.

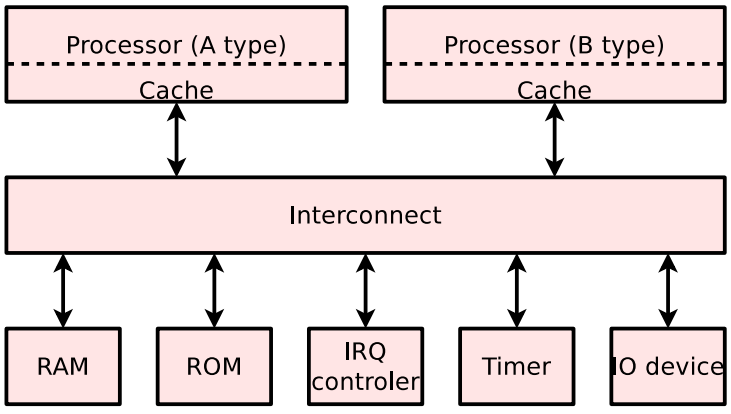

FIGURE 3.4 – Plateforme *SoCLib* de base utilisée pour le développement du noyau hétérogène

La figure 3.4 expose une plateforme multiprocesseur hétérogène simple utilisée pour la mise au point du noyau. Les processeurs d'architecture différente s'interfacent sur l'interconnect qui relie tous les composants de la plateforme. Les composants de *RAM* et de *ROM* permettent respectivement de stocker les données et le code du système.

On remarquera que les plateformes *SoCLib* utilisent une architecture moderne et des processeurs *RISC*. Du point de vue logiciel, elles sont très différentes des plateformes *Ibm-Pc*, basées sur une architecture rétrocompatible de plusieurs décennies et sur des processeurs *CISC*. Cette grande différence a été volontairement choisie pour contraindre la latitude de conception et mettre en défaut tout problème de non-généricité de la couche d'abstraction du matériel du noyau.

Pour garantir la portabilité, la mise au point du noyau s'est effectuée en premier lieu sur des plateformes non hétérogènes comme des stations de travail multiprocesseurs, des prototypes virtuels *SoCLib* à architecture homogène ou encore des cartes à base de microcontrôleurs.

3.3.1 Accès atomiques

Les accès atomiques à la mémoire sont généralement implémentés par des mécanismes relevant à la fois du processeur et de la plateforme. Au niveau processeur, il existe différents mécanismes liés à différentes approches :

– Les instructions de type *Read/Modify/Write* réalisent un accès atomique. Elles se basent sur la possibilité d'obtenir un accès exclusif à la mémoire. Seules les opérations prévues par le jeu d'instructions du processeur peuvent être réalisées.
– Les paires d'instructions de type *Linked Load* et *Store Conditional* (*LL/SC*) tentent une paire d'accès mémoire et testent l'atomicité effective de l'opération. Lorsque l'accès n'a pas pu être réalisé de manière atomique, l'écriture échoue et il faut relancer la séquence complète. Cette approche permet de réaliser n'importe quelle opération de manière atomique.

La première approche fonctionne sur les plateformes matérielles où les processeurs sont proches de la mémoire et où il est envisageable de bloquer l'accès à la mémoire à la demande d'un seul processeur. La seconde permet une plus grande souplesse car elle n'impose pas d'obtenir une exclusivité totale des accès à la mémoire en bloquant le reste de la plateforme.

La seconde approche, plus générique, est implémentée dans les processeurs *RISC* récents. Dans un système hétérogène, le même comportement est attendu par les différents processeurs vis-à-vis de la mémoire. C'est donc ce mécanisme plus souple et plus générique qui convient le mieux et qui est retenu pour le développement du noyau.

Les plateformes matérielles basées sur *SoCLib* permettent d'obtenir ce comportement unifié de type *LL/SC*.

On peut remarquer que certains processeurs ne proposant pas les instructions de lecture et écriture conditionnelles appropriées peuvent tout de même être exploités grâce à l'adjonction d'un automate câblé qui émule le comportement de cette approche pour les instructions d'accès atomiques.

3.4 Couche d'abstraction matérielle

La couche d'abstraction du matériel (*HAL*) implémentée dans le noyau capable d'hétérogénéité, offre une particularité par rapport aux *HAL* que l'on rencontre dans les autres noyaux : les effets engendrés par les appels à l'*API* doivent être identiques d'une architecture de processeur à l'autre.

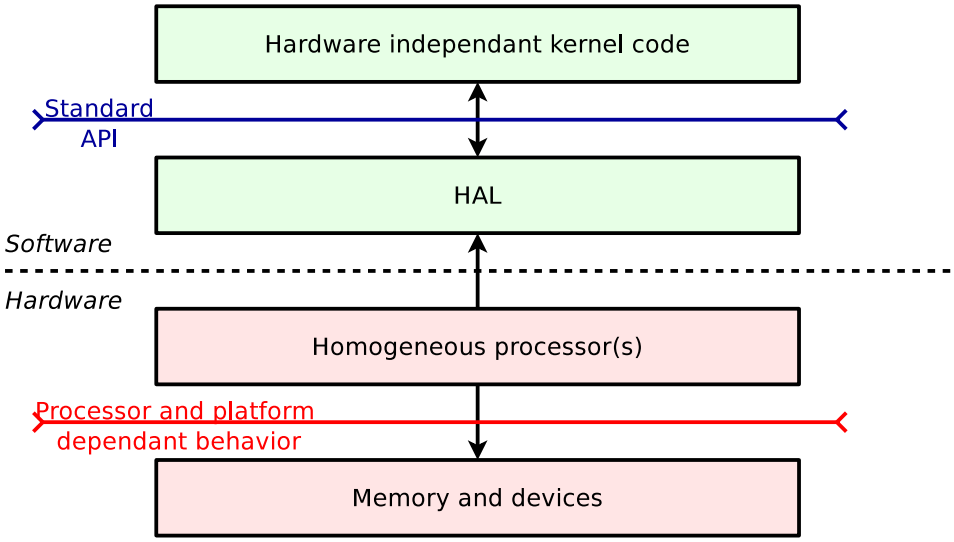

FIGURE 3.5 – Couche d'abstraction du matériel dans un noyau classique

En effet, la caractéristique d'une *HAL* classique (figure 3.5) est de garantir une abstraction logicielle, mais le détail des effets sur la plateforme peut varier d'un processeur à l'autre, pourvu que le service demandé au matériel soit satisfait. Dans un noyau classique, le code spécifique aux différentes architectures de processeurs sur lesquelles a été porté le noyau n'est jamais compilé et exécuté simultanément sur la même plateforme. Par ailleurs, le mécanisme matériel employé pour réaliser un accès atomique ou un accès aux périphériques importe peu, pourvu que le service soit rendu.

Dans le cas d'un système hétérogène la couche d'abstraction doit prendre soin d'opérer indifféremment, vis-à-vis de la plateforme, d'une architecture de processeur à l'autre. On a donc une contrainte double : sur l'interface logicielle et sur le comportement à l'égard du matériel. Un appel de fonction de la *HAL* doit donc générer le même effet de bord, quel que soit le type de processeur qui l'exécute. (figure 3.6)

Cet aspect complique sévèrement le développement de la *HAL* aussi un soin particulier doit être apporté à l'*API* et à l'architecture logicielle interne de cette couche. Une fois que la modélisation, garante de la généricité nécessaire, a été établie lors de l'élaboration du noyau, le portage de cette couche d'abstraction vers une nouvelle architecture de processeur reste relativement simple.

Les services suivants sont pris en charge par la couche d'abstraction du matériel :

– Démarrage et initialisation du système
– Invalidation et autres opérations sur les caches
– Accès atomiques à la mémoire
– Sérialisation des accès à la mémoire

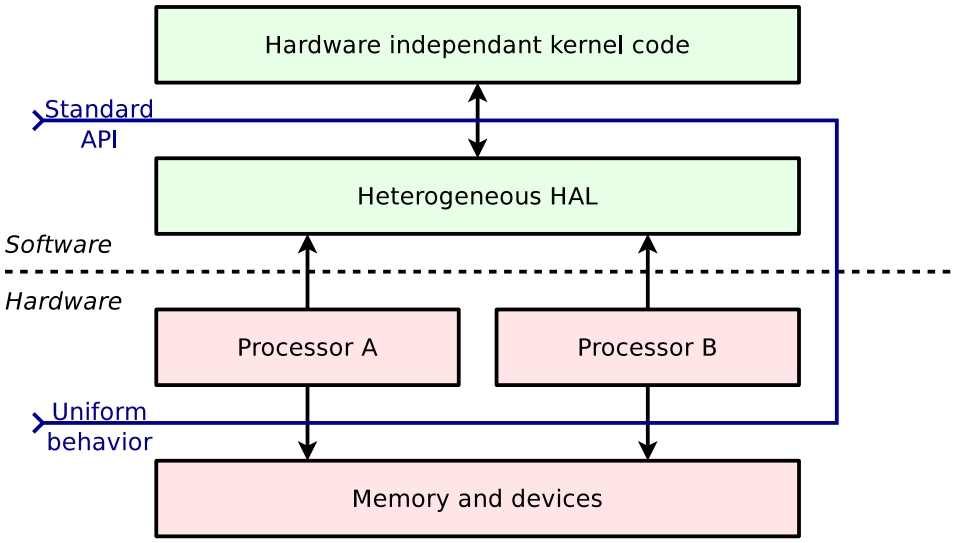

FIGURE 3.6 – Couche d'abstraction du matériel dans le noyau hétérogène

– Définition et opérations sur les verrous de type *spinlock*
– Accès mémoire non-alignés et d'endianness non native
– Accès à l'espace d'adressage des périphériques
– Variables globales du langage locales aux processeurs ou aux contextes d'exécution
– Opérations de gestion des pages de l'unité matérielle de mémoire virtuelle (*MMU*)
– Définition de types entiers portables
– Contextes d'exécution et opérations de permutation associées
– Opérations de gestion des interruptions, exceptions et appels système
– Gestion des interruptions inter-processeurs

Les sections suivantes abordent parmi ces services, ceux posant des problèmes spécifiques relatifs à l'hétérogénéité, et présentent les solutions apportées.

Il est important de noter que cette liste ne comporte que les services proposés par la *HAL*. D'autres services, tels l'allocation mémoire ou encore l'ordonnanceur de contextes d'exécution, font partie d'une couche supérieure du noyau du système d'exploitation ; celle-ci reste nécessaire à un système fonctionnel, mais est complètement abstraite de l'hétérogénéité.

La gestion de la mémoire virtuelle au travers d'une (*MMU*) a été étudiée et implémentée dans le noyau, dans le cadre d'autres travaux de recherche. Bien que la mémoire virtuelle soit fonctionnelle, elle n'a pas été utilisée dans les expérimentations sur l'hétérogénéité à ce jour.

3.4.1 Séparation processeur et plateforme

Parmi les services précédemment cités, certains ont une implémentation variant en fonction du type de processeur, d'autres dépendent uniquement des spécificités matérielles de la plateforme, d'autres enfin dépendent des deux aspects.

Etant donné que certains processeurs peuvent être employés dans plusieurs plateformes aux caractéristiques bien différentes, il semble naturel de distinguer le code spécifique aux processeurs et le

code spécifique aux plateformes prises en charge. Cette modularisation poussée du code n'existe toutefois pas dans les *HAL* de noyaux pourtant portés sur un grand nombre de plateformes, comme c'est le cas du noyau *Linux*.

Par exemple, les plateformes *SoCLib* et les plateformes monoprocesseurs simples à base de microcontrôleurs, contiennent des implémentations différentes mais toutes deux utilisables avec le module qui prend en charge les processeurs de type *ARM*.

Cette séparation qui n'est pas absurde dans le cas d'un noyau classique, prend tout son sens lorsque l'on souhaite prendre en charge l'hétérogénéité des processeurs de la plateforme. Il s'agit de distinguer le code qui ne sera compilé qu'une fois, de celui qui existera pour chaque instance de processeur. Ainsi, un seul module prenant en charge la plateforme sera sélectionné, alors que plusieurs modules spécifiques au processeur pourront l'être. On se rend compte ici que la plus forte contrainte de généricité et d'interchangeabilité du code, étudiée précédemment, s'applique surtout aux modules qui prennent en charge les différentes architectures de processeur.

3.4.2 Démarrage et initialisation du système

Lors du démarrage du système, comme dans tout noyau, les différents éléments matériels doivent être initialisés. Voici les divergences entre processeurs à considérer :

– En mode multiprocesseur, certaines architectures de processeurs élisent un processeur maître qui démarre seul tandis que les autres processeurs, alors en attente, doivent être réveillés explicitement par le processeur maître. Pour d'autres architectures, tous les processeurs sont démarrés en même temps, exécutent le même code de démarrage et doivent être synchronisés par voie logicielle.
– Certains processeurs complexes, tels les processeurs x86, comptent sur la mise en place d'une multitude de structures en mémoire auxquelles ils accèdent par des automates câblés. Ces structures concernent généralement les tables d'interruption ou encore les segments mémoire et doivent être mises en place très tôt pour disposer d'un processeur fonctionnel.

Afin de permettre une grande souplesse et de satisfaire aux contraintes d'initialisation des différents processeurs, le démarrage du système implique l'appel de toute une série de fonctions, tantôt spécifiques à la plateforme, tantôt spécifiques au processeur ; il se déroule ainsi :

– Le processeur exécute du code assembleur situé à l'adresse de démarrage (code spécifique au processeur) et saute très rapidement dans le code spécifique à la plateforme qui enchaîne les étapes suivantes du démarrage.
– Il existe un processeur maître par architecture de processeur ; le maître est désigné généralement suivant son identifiant numérique. Celui-ci est responsable d'effectuer toutes les éventuelles initialisations spécifiques à une architecture de processeur (1 seule fois pour la plateforme).
– L'allocateur mémoire est initialisé (code commun du système d'exploitation).
– Les structures globales en rapport avec les composants de virtualisation de la mémoire sont initialisées (code spécifique au processeur) si la virtualisation de la mémoire est activée.
– Chaque processeur appelle une fonction responsable des éventuelles initialisations devant être effectuées pour chaque processeur (code spécifique au processeur).
– Chaque processeur appelle une fonction responsable de l'initialisation de sa *MMU* si la virtualisation de la mémoire est activée (code spécifique au processeur).

– Tous les processeurs se mettent en attente active sur un verrou de type spinlock, sauf le processeur maître qui débloque ce verrou lorsqu'il arrive à cette étape.
– L'ordonnanceur de contextes d'exécution (*scheduler*) est initialisé (code commun du noyau).
– La main est donnée aux couches plus hautes du système d'exploitation qui vont initialiser les périphériques puis lancer l'application.

On remarque donc que les problèmes présentés sont pris en charge de la manière suivante :

– Le scénario du processeur maître qui démarre seul est repris et imposé grâce à un verrou, y compris pour les architectures de processeur qui ne se comportent pas de cette manière nativement.
– Les différentes étapes de l'initialisation sont découpées finement et correspondent à différents appels de fonctions définis dans l'*API*, ayant chacun leur rôle bien précis, permettant à chaque implémentation d'un module processeur d'effectuer ou non des actions lors de chaque étape.

3.4.3 Accès atomiques et verrous

La mise en place des accès atomiques et celle des verrous de type *spinlock* vont de pair. Leur implémentation dans la couche d'abstraction soulève un problème intéressant qui les rend interdépendants. L'un peut dépendre de l'autre, ou vice versa, selon les primitives disponibles au niveau matériel.

En effet, les verrous peuvent être implémentés grâce aux opérations atomiques dans les plateformes où la mémoire peut être accédée de manière atomique, tandis que les opérations atomiques peuvent être réalisées en utilisant des verrous dans les plateformes comportant des composants de verrous matériels spécifiques.

Etant donnée la séparation complète du code spécifique au processeur d'une part et à la plateforme d'autre part, la gestion des opérations atomiques d'accès mémoire du processeur se situe dans un module différent de celui du support des verrous au niveau plateforme.

En fonction des ressources matérielles disponibles, il convient donc d'organiser cette interdépendance de façon à fournir les deux services de manière transparente à la couche supérieure.

La compilation conditionnelle et une API interne bien définie permettent de résoudre ce problème d'interdépendance.

Verrous matériels

Lorsque seuls sont disponibles les verrous matériels, la couche d'abstraction compte sur la présence d'un périphérique permettant d'associer un identifiant unique de verrou à un jeu d'opérations classiques de prise et de libération.

Instructions d'accès atomiques

Lorsque les instructions d'accès atomiques sont disponibles dans les processeurs utilisés, leur usage est préféré car elles permettent la déclaration et l'initialisation statique de variables globales supportant les accès atomiques au milieu des autres données. Ceci est un prérequis pour l'implémentation complète de certaines bibliothèques comme les *thread*s *POSIX*.

3.4.4 Accès aux périphériques

L'accès aux registres des périphériques s'effectue au travers d'un espace d'adressage. Cet espace d'adressage est exploité par les pilotes de périphériques qui font partie des couches plus hautes du système.

Beaucoup d'architectures de processeur ne distinguent pas l'espace d'adressage destiné à la mémoire, de celui réservé aux périphériques. Il existe cependant des architectures dotées d'un espace d'adressage spécifique :

– Les processeurs *Sparc* disposent d'instructions d'accès mémoire permettant de désigner l'espace d'adressage dans lequel doit s'effectuer l'accès.
– Les processeurs *x86* disposent d'instructions d'accès spécifiques à un espace d'adressage réservé exclusivement aux périphériques. Il faut noter que cet espace d'adressage est restreint en taille ; certains périphériques ne l'emploient pas et restent accessibles par l'espace d'adressage mémoire. D'autres l'emploient de manière exclusive.
– Certains microcontrôleurs disposent d'un espace spécifique pour l'accès aux périphériques, soit par conception, soit par les mécanismes de multiplexage mis en œuvre dans les plateformes qui les emploient.

Certains processeurs ou certaines plateformes nécessitent l'ajout d'instructions de synchronisation mémoire ou de contrôle de cache pour s'assurer que l'accès vers le périphérique n'est pas temporisé ou désordonné.

Ce constat donne lieu à la mise en place d'une API permettant l'accès à deux espaces d'adressage : l'espace mémoire et l'espace IO, qui sont confondus pour les processeurs n'en supportant qu'un.

Ce service permet de développer des pilotes de périphériques portables et indépendants des spécificités de la plateforme, dans une certaine mesure.

3.4.5 Variables globales contextuelles

Lors du développement d'un noyau complexe, la modélisation et la conception modulaire du code ne peuvent pas être laissées de coté. Le langage et les outils de développement doivent apporter la flexibilité et les fonctionnalités nécessaires permettant de résoudre les problèmes courants de manière efficace et élégante.

Il est une fonctionnalité qui apparaît incontournable, mise en place dans tous les environnements de développement parallèles : c'est la possibilité de déclarer simplement une variable globale du langage qui reste locale à un contexte d'exécution.

Cette fonctionnalité est par exemple déjà proposée par le mot clef `__thread` de la bibliothèque de *thread*s *POSIX* et des compilateurs associés.

La version de cette approche proposée par la couche d'abstraction dont il est question ici va un peu plus loin encore puisqu'elle permet deux modes de localité :

– La déclaration de variables globales du langage, locales au contexte d'exécution.
– La déclaration de variables globales du langage, locales au processeur.

Il est possible, grâce à cette fonctionnalité, de déclarer des variables simplement (figure 3.7), là où une procédure plus complexe d'allocation de structures aurait été nécessaire pour chaque processeur lors de l'initialisation du noyau.

```
CPU_LOCAL interrupt_hanlder_t *interrupt_hanlder;
CONTEXT_LOCAL uint8_t context_stack[STACK_SIZE];
```

FIGURE 3.7 – Exemple de variables globales contextuelles

Ces deux fonctionnalités sont employées dans de nombreuses autres parties du système, aussi bien dans la couche d'abstraction que dans les parties supérieures du noyau, et restent également disponibles pour l'application en mode noyau.

Dans la solution proposée, cette fonctionnalité exploite deux mécanismes :

– La possibilité de placement des symboles du langage C dans une section déterminée du fichier binaire de sortie grâce à un attribut dédié du compilateur. Ceci permet de placer une variable contextuelle dans une section spéciale.
– L'utilisation de registres spécifiques du processeur qui pointent vers les sections adéquates en mémoire, préalablement allouées et initialisées par la *HAL* pour chaque contexte.

3.4.6 Les types entiers

Le problème de la largeur des types entiers, évoqué précédemment, trouve une solution dans l'utilisation des types standards définis par le standard *C99*.

La définition de ces types est fournie par la couche d'abstraction. Les types suivants sont définis par compilation conditionnelle et configuration des sources, selon le standard :

– Les types permettant la déclaration d'un entier avec une largeur bien définie : `uint`N`_t` et `int`N`_t`.
– Les types permettant la déclaration d'un entier avec une largeur au moins égale à une certaine valeur : `uint_fast`N`_t` et `int_fast`N`_t`.
– Les types permettant la déclaration d'un entier de même largeur qu'un pointeur : `uintptr_t` et `intptr_t`.

3.4.7 Les événements

Les événements modifiant le fil d'exécution d'une application sont : les interruptions, les exceptions et les appels système.

Tous les processeurs offrent un mécanisme permettant l'exécution de code prédéterminé lorsque ceux-ci surviennent. Cependant ces mécanismes diffèrent et la couche d'abstraction doit proposer une interface uniforme pour permettre le traitement de ces événements par les couches supérieures du noyau.

Une fois que le code spécifique au processeur a récupéré le contrôle, il peut transférer celui-ci à une routine enregistrée par les couches supérieures via l'*API* générique proposée.

L'hétérogénéité de la plateforme peut nécessiter l'utilisation de routines différentes pour les différents types de processeurs. De même, l'implémentation de différentes interfaces de systèmes d'exploitation, au-dessus de la couche d'abstraction, nécessite la cohabitation de différentes routines d'appels système.

Les pointeurs internes vers les routines enregistrées dynamiquement, exploitent donc les mécanismes de variables globales contextuelles présentées précédemment. Ainsi :

– Les routines d'exceptions et d'interruptions sont affectées séparément pour chaque processeur.
– Les routines d'appels système sont affectées séparément pour chaque contexte d'exécution.

3.5 Génération des fichiers binaires

La génération des fichiers binaires, propre à l'exécution sur une plateforme matérielle de l'ensemble logiciel noyau et application, constitue ce que l'on appelle communément le *build process*. Celui-ci se compose de différentes étapes : configuration, compilation et édition de liens, présentées par la suite.

Dans le cas d'un noyau hétérogène comme celui dont il est question ici, ce processus de génération des binaires est plus complexe que dans un système classique. Plusieurs points le différencient :

– Plusieurs compilations des mêmes sources doivent avoir lieu (une pour chaque type de processeur cible).
– La configuration des sources, en amont de la compilation, doit rester cohérente pour chaque type de processeur vis-à-vis des autres processeurs et des caractéristiques choisies pour la plateforme. C'est-à-dire qu'aucune structure de données ne doit être interprétée différemment en raison de directives de compilation conditionnelle.
– L'édition des liens, après la compilation, décide du placement des symboles dans les sections et doit donc prendre soin de générer des binaires où les symboles sont aux mêmes adresses pour tous les processeurs.

Ajoutons comme contrainte que les plateformes classiques, non hétérogènes, doivent être prises en charge sans complexité supplémentaire par rapport à un autre noyau.

3.5.1 Configuration des sources

La configuration du code source doit permettre de définir plusieurs caractéristiques de la plateforme matérielle :

– L'architecture des processeurs cibles.
– La largeur des types scalaires à adopter, permettant une harmonisation éventuelle entre architectures de processeur en présence (telle qu'elle a été éxpliquée antérieurement).
– Les différentes fonctionnalités et modules du noyau à activer ainsi que leur configuration.
– Les pilotes de périphériques à compiler.
– Les options de compilation à destination du compilateur concernant l'optimisation et la génération du code machine.

Etant donné le nombre très important d'options qui peuvent être ajustées, il convient de mettre en place un outil permettant de vérifier la cohérence entre les différents choix dans la configuration des sources.

Cet outil doit prendre en compte la spécification des différents paramètres sur lesquels on peut intervenir. Ces paramètres sont définis par leur nom et par les différentes contraintes qui les lient entre eux, de telle sorte que l'on puisse détecter toute configuration invalide. Cette spécification fait partie du code source.

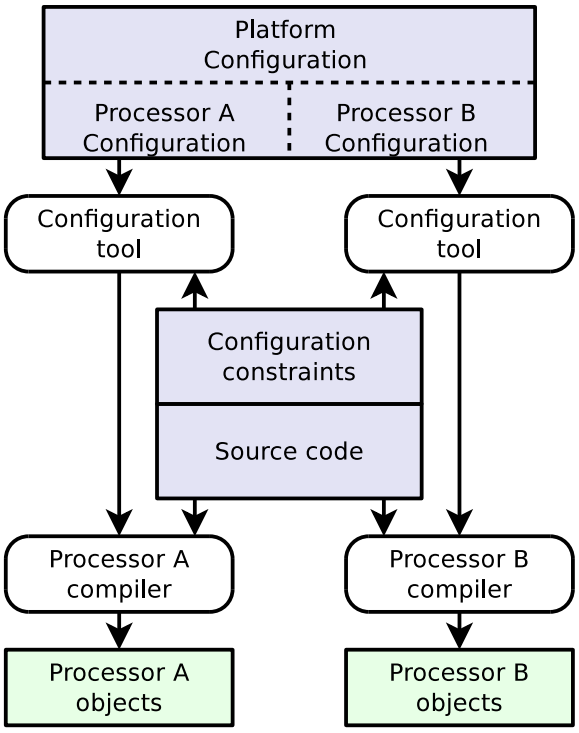

FIGURE 3.8 – Configuration et compilation pour plateforme hétérogène

Lors de la mise en œuvre du code source et du déploiement d'applications sur une plateforme matérielle, l'utilisateur du noyau doit fournir une configuration valable, en ajustant les valeurs par défaut proposées pour les différents paramètres.

Dans le cas d'un système hétérogène chaque architecture de processeur présent dans la plateforme doit disposer de sa configuration. Le détail du fonctionnement du système de configuration proposé est présenté plus loin, dans la section 4.1.2.

3.5.2 Compilation

On s'intéresse ici à la génération du code machine à partir du code source. Cette étape comprend en fait la compilation du code C en code assembleur puis la génération des fichiers objets.

Rappelons que la génération du code binaire, pour chaque architecture de processeur choisie, doit respecter certaines contraintes. La compilation doit assurer une certaine partie de l'uniformité du code ; ceci implique les points suivants :

– Certains aspects du code généré à la compilation, notamment le format des données en mémoire, ne répondent à aucun standard ou sont mal spécifiés et sont laissés à la discrétion du compilateur. Le compilateur utilisé doit être le même pour les différents processeurs cibles.
– Les options de compilation doivent être choisies pour assurer l'uniformité des tailles de types de données, et de l'alignement des champs dans les structures.

La compilation ne permet cependant pas de contrôler les adresses mémoire affectées aux différentes fonctions et variables en mémoire. L'ordre d'émission des symboles par le compilateur n'ayant aucune importance dans le cas des systèmes classiques, c'est un point qui n'est pas pris en considération par les compilateurs. Il apparaît que pour un même compilateur, l'aléa introduit par les différentes structures de données internes, comme les tables de hashage, ne permet pas d'obtenir des résultats prédictibles sur ce point. Il faut donc résoudre ce problème plus tard, lors de la phase d'édition des liens.

3.5.3 Edition des liens

L'édition des liens est une étape importante dans la génération des binaires hétérogènes puisqu'elle doit permettre de corriger l'ordre de rangement et les adresses mémoire de chaque symbole pour obtenir l'uniformité nécessaire.

Les objets binaires

Les fichiers binaires obtenus par l'étape de compilation/assemblage du code source contiennent les codes binaires des instructions spécifiques au jeu d'instructions du processeur cible.

Bien qu'il s'agisse de code binaire, celui-ci n'est toutefois pas encore exécutable. Toutes les références vers les variables globales et fonctions ne sont pas encore connues et certaines instructions restent à compléter pour pouvoir être exécutées.

En effet, chaque fichier source compilé engendre un fichier objet qui exploite généralement les fonctions situées dans d'autres fichiers source. Le code binaire exécutable d'un symbole reste donc à compléter par l'éditeur de liens. Les fichiers objets contiennent toutes les structures nécessaires à ces opérations de modification ultérieure des références :

– L'organisation en sections permet de séparer le code des données préallouées que sont les variables globales.
– Ces fonctions et variables constituent des symboles qui sont nommés. Le nom d'un symbole l'identifie de manière unique. La table des symboles dans le fichier objet contient une entrée pour chaque symbole qui est : soit défini, soit requis par le fichier source correspondant.
– Des tables de relocations gardent une trace de chaque instruction laissée incomplète et qui devra être modifiée ultérieurement (figure 3.9).

Cette caractéristique de code incomplet des fichiers objets permet de résoudre deux problèmes dans le cas d'une compilation classique, et davantage, dans le cas d'une compilation hétérogène :

– Les adresses des symboles peuvent rester inconnues jusqu'à la mise en commun des différents fichiers objets d'un projet.

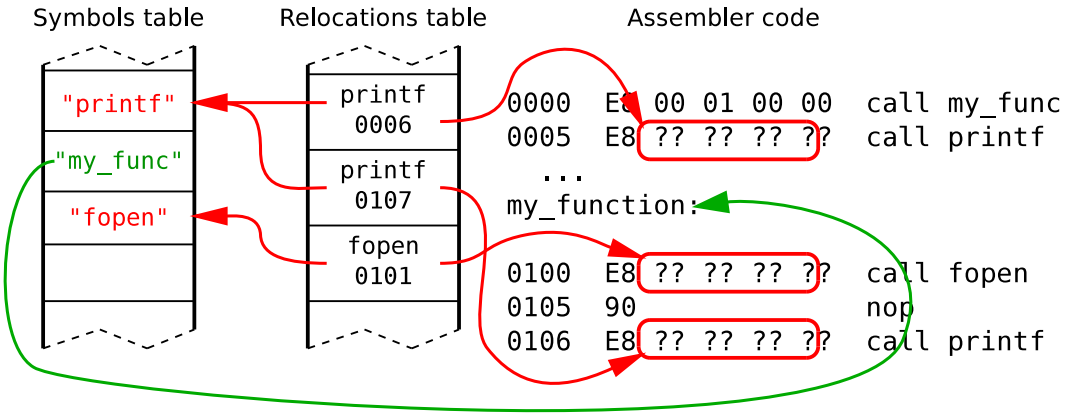

FIGURE 3.9 – Table de symboles et table de relocation d'un objet

– Les adresses des symboles peuvent être décalées lors de la concaténation des sections ; ceci survient également lors de la mise en commun de fichiers objets (figure 3.10).
– Les adresses des symboles peuvent encore évoluer pour être réordonnées de manière tardive et ainsi subir l'uniformisation des fichiers objets, nécessaire à l'hétérogénéité.

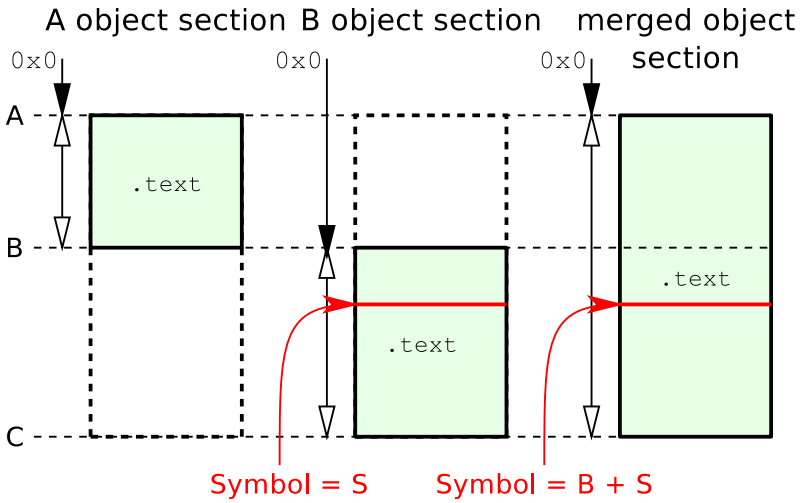

FIGURE 3.10 – Changement d'adresse d'un symbole lors de la fusion de deux objets

Réorganisation des symboles

Il est donc question de réorganiser complètement les sections de données et de code. Une étape supplémentaire est introduite pour réaliser cette opération dans le cas d'un système hétérogène. Cette étape survient après la compilation et avant l'édition des liens finale habituelle qui génère le binaire exécutable pour chaque architecture de processeur.

Quand on a obtenu des sources l'ensemble des fichiers objets, et que l'on a pu les fusionner en un seul fichier objet, un outil supplémentaire entre en jeu. Il reçoit un fichier objet contenant l'ensemble du système pour chaque type de processeur et effectue les opérations suivantes :

– Mise en place d'une liste des symboles communs aux différentes compilations hétérogènes, à partir de leur nom.
– Affectation d'une adresse unique à chaque symbole
– Réorganisation effective de tous les symboles dans les fichiers objets propres à chaque architecture, selon cette liste commune de symboles. Il en découle que tous les symboles possédant un nom identique sont relogés à la même adresse.
  Lorsqu'un symbole possède un nom unique qui ne se retrouve pas dans les objets des autres architectures cibles, il lui est également alloué un espace propre, de telle sorte qu'il reste possible de déclarer des fonctions ou variables spécifiques à une architecture de processeur.
– Ecriture de la version réorganisée de l'objet, destinée à l'édition des liens classique.

Cette réorganisation diffère légèrement selon le type de section :

– Pour les sections de code, les différentes instances compilées d'une même fonction ne possèdent pas nécessairement la même taille d'un jeu d'instructions à l'autre. La taille de la plus grande fonction est considérée.
– Pour les sections de données, les symboles de même nom doivent posséder strictement la même taille.

Cette étape de réorganisation nécessaire implique des transformations complexes dans le fichier objet pour prendre en compte tous les éléments qui le constituent : sections, symboles, relocations... Ces transformations sont exposées plus loin.

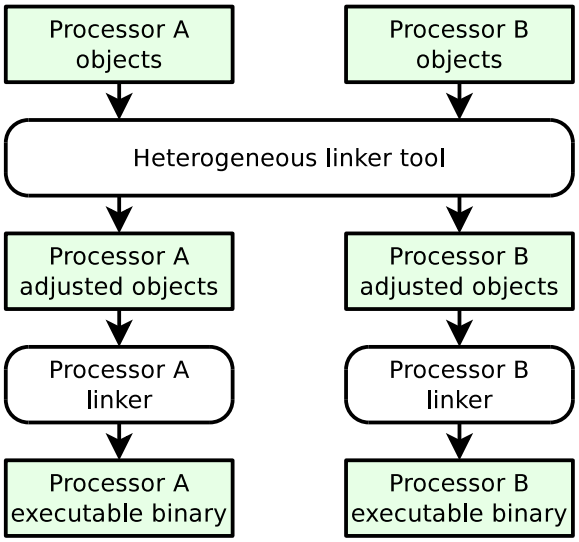

FIGURE 3.11 – Edition des liens pour plateforme hétérogène

Une fois la réorganisation du fichier terminée, il devient possible de générer les exécutables. Cette étape fige définitivement le placement des fonctions et des variables dans les sections, par la résolution de toutes les relocations ; c'est-à-dire que toutes les adresses sont inscrites définitivement dans le code binaire des instructions qui restaient incomplètes jusqu'ici. Le code est maintenant exécutable par le processeur.

L'étape supplémentaire, introduite dans le processus de construction des fichiers binaires à partir des sources, nécessite l'utilisation d'un outil spécifique mais reste indépendante des opérations de

compilation et d'assemblage effectuées en amont, et de l'édition des liens finale effectuée en aval par des outils standards. De cette façon, une compilation pour une plateforme classique non-hétérogène ne nécessite que des outils standards.

Pour les expérimentations nous utilisons le compilateur C *GNU* et le package *binutils* associé, sans y apporter de modifications. Le format de fichier binaire retenu est le très répandu format *ELF* (*Executable and Loadable file Format*).

3.6 Conclusion

La réalisation d'un noyau rendant possible l'exécution d'un système d'exploitation sur une plateforme matérielle multiprocesseur hétérogène est envisageable dans la plupart des cas.

Le bénéfice apporté par un tel système est réel puisqu'il permet l'exploitation de processeurs aux spécialisations et complexités différentes. Il faut toutefois éviter les processeurs traitant des largeurs ou endianness de mots différentes, mais cette contrainte n'est pas plus forte que celle qui dirige habituellement le choix des périphériques.

L'architecture en exo-noyau retenue permet la plus grande flexibilité quant aux utilisations diverses qui pourront être faites du noyau.

Le premier obstacle à la réalisation concerne l'hétérogénéité des codes binaires exécutables, aux jeux d'instructions différents ou identiques. Il peut être résolu grâce à la virtualisation de la mémoire de code et à la réorganisation des fonctions et variables dans les sections chargées en mémoire. Ceci permet de masquer élégamment cette différence sans perte de performances à l'exécution.

Le second problème concerne les mécanismes matériels système, qui peuvent différer d'un processeur à l'autre, de par la mise en œuvre logicielle requise et de par leurs comportements vis-à-vis de la plateforme. Il peut être résolu grâce à une couche d'abstraction du matériel adaptée. Cette *HAL*, bien que spécialement conçue pour les plateformes hétérogènes, n'implique pas de coût en performances par rapport aux autres *HAL* classiques, étant donné qu'il s'agit principalement de soigner la modélisation interne et la modularité de son code source.

Certains composants de la plateforme matérielle devront, quant à eux, supporter l'hétérogénéité en proposant des mécanismes d'accès atomiques utilisables par tous les processeurs en présence, et permettre la virtualisation de la mémoire contenant les codes exécutables.

Il est intéressant de noter que rien n'empêche un tel noyau d'être exploité sur une plateforme non hétérogène, avec la même facilité de mise en œuvre et le même niveau de performances qu'un noyau classique.

Chapitre 4

Réalisation du noyau

Le noyau de système d'exploitation, dont le développement a été initié dans le cadre de cette thèse, se nomme *MutekH*. Il est composé de plusieurs modules dont certains sont spécifiques au projet comme l'API d'abstraction du matériel (plateforme et processeurs) nommée *Hexo*. La plupart des autres modules sont des implémentations de services standardisés et sont nommés en conséquence.

Cette section présente les différents développements noyau qui permettent le support natif de l'hétérogénéité dans la solution proposée, ainsi que quelques unes des fonctionnalités importantes de *MutekH* ; sans rapport direct avec l'hétérogénéité, celles-ci ont toutefois facilité la réalisation et/ou assuré la portabilité, la modularité et la généricité du code requises par le support de l'hétérogénéité.

4.1 Modularité

Effectivement, la modularité de la solution mise en place, la portabilité et la généricité du code occupent une place importante dans la réalisation d'un noyau de système d'exploitation capable de supporter nativement l'hétérogénéité.

Ceci est d'autant plus vrai qu'on souhaite exploiter cette fonctionnalité de manière transparente, et proposer différentes API et bibliothèques, pour supporter une vaste gamme d'applications parallèles déjà existantes.

Une vue d'ensemble de l'architecture logicielle de *MutekH* est présentée figure 4.1.

4.1.1 Organisation

Le code source du noyau est organisé en modules classables par catégorie. On peut établir ce classement en donnant quelques exemples de modules associés parmi ceux existants :

– Les modules offrant des services purement algorithmiques et indépendants, comme les conteneurs génériques fournis par `gpct`, ou le sous-ensemble du module `libc` proposant les opérations standards sur les chaînes.

– La couche d'abstraction du matériel composée du module principal `hexo` et des modules spécifiques à une plateforme ou à un processeur situés dans `arch` et `cpu`.
– Le code de l'exo-noyau qui n'est pas directement dépendant du matériel, notamment les différents allocateurs mémoire ou l'ordonnanceur, regroupés dans le module `mutek`.
– Les pilotes de périphériques situés dans le module `drivers`.
– Les nombreuses bibliothèques implémentant les API de haut niveau utilisées par les applications pour gérer leurs contextes d'exécution et le parallélisme. C'est le cas, entre autres, de la bibliothèque de *thread*s *POSIX* (`libpthread`) ou de la bibliothèque *OpenMP* (`libgomp`)... Cette catégorie inclut également les bibliothèques qui mettent en place un espace utilisateur avec des appels système comme une implémentation d'*UNIX*. La modularisation des différents services et API est une caractéristique déterminante de l'architecture en exo-noyau.
– Les nombreuses bibliothèques de services rencontrées généralement dans les noyaux, comme la gestion du système de fichiers (`libvfs`) ou encore la pile réseau (`libnetwork`)...

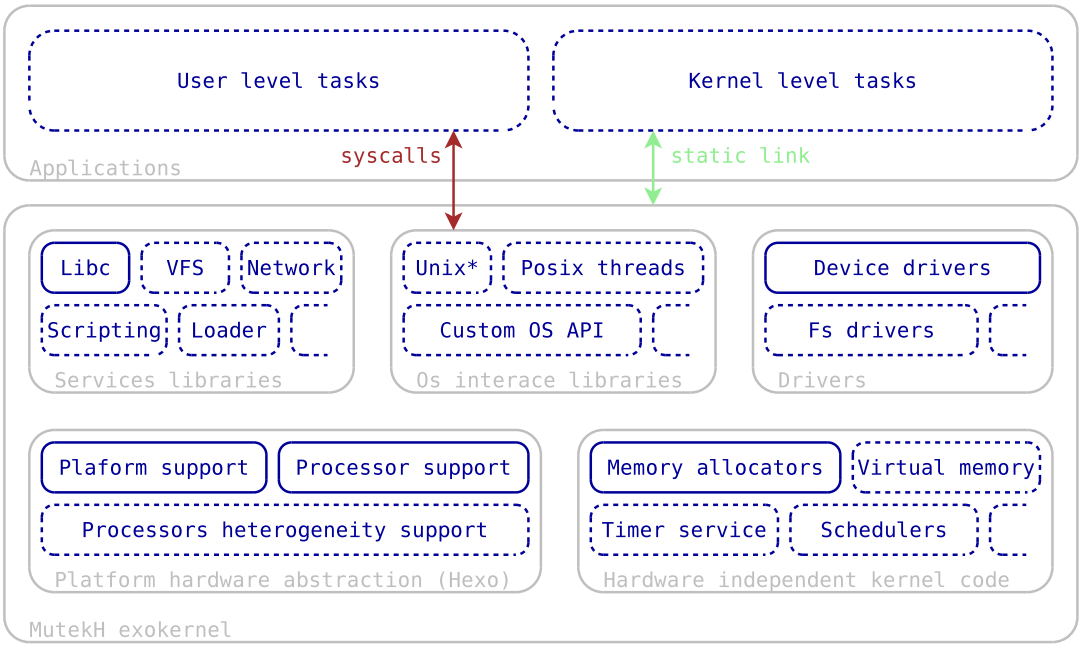

FIGURE 4.1 – Vue globale de l'architecture logicielle de MutekH

4.1.2 Configuration des sources

Outre le fait que la configurabilité du code source est une caractéristique partagée par tous les projets de noyaux de systèmes d'exploitation conséquents, elle va de pair avec la modularité et permet :

– l'adaptation du code source du noyau à la plateforme et à l'application.
– le contrôle des diverses contraintes de dépendance et d'exclusivité entre les fonctionnalités et les modules, évidentes pour le développeur du noyau mais pas forcément pour le développeur d'applications.

Dans le cadre de notre objectif, la configurabilité des sources permet de réaliser correctement les différentes compilations afin de cibler les différents processeurs d'une plateforme hétérogène. Nous détaillons ici les mécanismes de configuration présentés dans la section 3.5.1 et illustrons leur mise en œuvre.

Un outil capable de gérer la configuration des sources de manière poussée permet de :

– garder un contrôle fin des différences entre les processeurs.
– garantir que seul le code spécifique aux processeurs diffère lors d'une compilation pour plateforme hétérogène.

L'outil de configuration de *MutekH* se présente sous la forme d'un script en langage *perl* sans dépendance externe.

Les différents fichiers de configuration

Ce système de configuration repose sur deux types de fichiers de configuration :

```
%config CONFIG_PTHREAD
  desc Enable POSIX thread support
  flags auto
  depend CONFIG_MUTEK_SCHEDULER
  module libpthread PThread library
%config end

%config CONFIG_PTHREAD_CHECK
  desc Enable extensive error checking for pthread API
  when CONFIG_DEBUG
  parent CONFIG_PTHREAD
%config end

%config CONFIG_PTHREAD_RWLOCK
  flags auto
  desc Enable pthread_rwlock_* primitives
  parent CONFIG_PTHREAD
  depend CONFIG_MUTEK_RWLOCK
%config end

...
```

FIGURE 4.2 – Exemple de définition de jetons de configuration de la bibliothèque de *thread*s *POSIX*

– Les fichiers décrivant la configurabilité : Ces fichiers font partie du code source du projet, ils définissent des jetons de configuration pour chaque fonctionnalité et chaque paramètre qui peuvent être ajustés, ainsi que les valeurs par défaut et les contraintes et règles qui les lient. Ces fichiers ne définissent pas une configuration particulière du noyau, mais déterminent plutôt l'ensemble des configurations possibles. Plusieurs centaines de jetons sont définies dans *MutetkH*. La figure 4.2 montre un échantillon de ces définitions pour la bibliothèque `libpthread` : Ici nous définissons le jeton `CONFIG_PTHREAD` activant le module `libpthread` et dépendant de l'ordonnanceur. Les fonctionnalités de vérification et de verrous en lecture/écriture du module, sont représentées par les jetons `CONFIG_PTHREAD_CHECK` et `CONFIG_PTHREAD_RWLOCK`, dépendant eux-mêmes d'autres fonctionnalités du noyau.

– Les fichiers de configuration de construction des binaires : Ces fichiers permettent de définir la configuration précise du système d'exploitation à utiliser pour une application donnée et pour une plateforme donnée. La figure 4.3 montre un exemple de fichier de configuration pour une application simple. Dans cet exemple, l'application *hello* utilise la bibliothèque de *thread*s *POSIX*, les sémaphores de la bibliothèque C et la bibliothèque mathématique standard. Les paramètres relatifs à la plateforme sont pris en charge par les fichiers inclus.

```
# set binary file name
%set OUTPUT_NAME hello

# Application license
CONFIG_LICENSE_APP_LGPL

# Used libraries
CONFIG_PTHREAD
CONFIG_LIBC_SEMAPHORE
CONFIG_LIBM

# Platform build configuration files
%include ../common/build_options.conf
%include ../common/platforms.conf
```

FIGURE 4.3 – Exemple de configuration d'une application simple

Les fichiers de configuration, paramétrant une application ou une plateforme donnée, peuvent être une simple liste de jetons de configuration à activer mais ils peuvent également inclure d'autres fichiers et comporter plusieurs sections sélectionnées conditionnellement par l'utilisateur à la compilation. L'inclusion des fichiers permet de déporter la configuration spécifique à une plateforme, hors du fichier de configuration d'une application.

Après validation selon les contraintes énoncées dans les fichiers de configurabilité, la configuration retenue pour la construction des sources est enrichie de valeurs par défaut puis exportée pour intervenir à plusieurs niveaux : dans le code source, via la compilation conditionnelle du préprocesseur C, et dans le système de construction, via les directives conditionnelles de *GNU make*.

La configuration d'une plateforme hétérogène

Outre le fait qu'elle permet à l'utilisateur de compiler aisément une même application pour plusieurs plateformes, l'activation conditionnelle de sections dans le fichier de configuration, permet de traiter le problème des différences liées aux processeurs lors d'une compilation hétérogène. Le processus de compilation et son invocation sont abordés par la suite, section 4.1.3.

La configuration présentée figure 4.4 montre comment quelques paramètres peuvent différer au sein d'une large configuration commune de plateforme hétérogène. Dans ce cas les processeurs retenus pour la génération de binaires hétérogènes sont de type *Mips32* et *ARM* et l'adresse de la section de données en lecture seule `.rodata` n'est pas localisée à la même adresse en mémoire partagée suivant le processeur (voir section 5.2.5). Selon les sections considérées lors de la compilation, seule la sous-section `soclib-arm` ou `soclib-mips32el` sera prise en compte.

```
...

%subsection pf-het
  %types platform

  CONFIG_HET_BUILD

  # Memory layout
  CONFIG_RAM_ADDR 0x7f000000
  CONFIG_RAM_SIZE 0x01000000
  CONFIG_ROM_SIZE 0x00100000
  CONFIG_HETROM_ADDR 0x60000000
  CONFIG_HETROM_SIZE 0x00100000

  %subsection soclib-arm
    %types supported_het_cpu
    CONFIG_ROM_ADDR 0x80100000
    CONFIG_CPU_ARM_SOCLIB
  %end

  %subsection soclib-mips32el
    %types supported_het_cpu
    CONFIG_ROM_ADDR 0x80000000
    CONFIG_CPU_MIPS32EL_SOCLIB
  %end
%end

...
```

FIGURE 4.4 – Configuration d'une plateforme hétérogène SoCLib

4.1.3 Processus de construction des fichiers binaires

La construction des binaires inclut les étapes de compilation, d'assemblage et d'édition des liens présentées dans les sections 3.5.2 et 3.5.3.

Lors de la compilation du noyau et d'une application pour une plateforme classique (sans hétérogénéité), le processus de construction est le suivant :

– Lecture des fichiers de configurabilité dans le code source du noyau.
– Lecture de la configuration retenue pour l'application et la plateforme.
– Calcul des dépendances, précompilation conditionnelle, compilation et assemblage du code source des modules retenus dans la configuration.
– Edition des liens entre les fichiers objets pour produire le binaire exécutable final.

La compilation est invoquée par l'utilisateur en spécifiant le fichier de configuration de l'application, en précisant éventuellement les sections du fichier de configuration à considérer. La figure 4.5 donne deux exemples d'invocation du processus de construction de *MutekH*, le premier pour un fichier de configuration simple (à plat) et le second précisant la nécessité d'activer les sections de configuration ciblant la plateforme `tutorial` de type *SoCLib* qui emploie un processeur *Mips 32*.

La compilation d'une plateforme hétérogène enrichit un peu ce processus en l'exécutant en parallèle pour les différents processeurs cibles, avant de réaliser l'uniformisation, présentée précédemment

```
$ make CONF=examples/hello/config_x86
$ make CONF=examples/hello/config BUILD=soclib-mips32el:pf-tutorial
```

FIGURE 4.5 – Exemples d'invocation de construction de *MutekH* sans hétérogénéité

dans la section 3.5.3 :

– Lecture des fichiers de configurabilité dans le code source du noyau.
– Lecture de la configuration retenue pour l'application et la plateforme et différenciation pour chaque type de processeur.
– Calcul des dépendances, précompilation conditionnelle, compilation et assemblage du code source des modules retenus dans la configuration (étape répétée pour chaque type de processeur).
– Edition des liens entre les fichiers objets afin de produire un unique fichier objet pour chaque processeur cible.
– Réorganisation des symboles dans les fichiers objets obtenus pour uniformiser les binaires destinés aux différents processeurs.
– Edition des liens afin de transformer chaque fichier objet en un binaire exécutable final.

La compilation est invoquée par l'utilisateur en spécifiant simplement le fichier de configuration de l'application ainsi que la liste des processeurs ciblés, comme l'illustre la figure 4.6 déjà présentée.

```
$ make CONF=examples/hello_het/config BUILD=pf-het \
       EACH=soclib-mips32el:soclib-arm
```

FIGURE 4.6 – Exemple d'invocation de construction de *MutekH* hétérogène

L'étape de réorganisation des symboles dans les fichiers objets, proposée dans la section 3.5.3, est complexe ; elle a nécessité le développement d'une bibliothèque de manipulation des fichiers *ELF* et de l'outil spécifique dédié à cette étape. Cette réalisation fait l'objet du chapitre 5.

4.2 Les modules essentiels

Parmi tous les modules présents dans *MutekH*, certains occupent une place centrale de par les services de base qu'ils fournissent ; ils sont utilisés par tous les autres modules.

4.2.1 La bibliothèque de structures de données

Un noyau comme *MutekH* emploie en grand nombre, au travers des différents modules, les structures de données classiques comme les listes chaînées, les tableaux à redimensionnement dynamique, les tables de hashage... ainsi que des mécanismes objets tels que le comptage de références et la destruction automatique. En ce qui concerne les services de base du noyau, on peut citer les files d'attente et d'exécution de l'ordonnanceur, ou encore le chaînage des blocs de l'allocateur mémoire.

Le langage C ne fournit pas en standard de bibliothèques de conteneurs du type de la STL (*standard template library*) du langage C++ par exemple. Le choix du langage C reste cependant intéressant

pour les développements où on souhaite rester proche du système et garder une maîtrise relative du code généré. Bien que le langage C soit un sous-ensemble du C++ et que certains décident de développer des noyaux en C++ afin de profiter de quelques extensions du langage seulement, ce choix ouvre la porte à des dérives qui conduisent à l'utilisation de constructions de plus en plus complexes avec le temps et les contributions. Le risque est la perte de contrôle sur la complexité du code généré.

L'utilisation d'une bibliothèque de conteneurs pour le langage C permet donc de bénéficier des avantages de factorisation de ces algorithmes et structures de données indispensables, tout en limitant l'introduction de bogues courants.

La bibliothèque retenue est fournie par le module `gpct`, développée parallèlement au noyau et utilisée dans d'autres projets. Elle a la particularité d'être une bibliothèque *template*, c'est-à-dire qu'elle génère des fonctions C adaptées et spécifiques à chaque structure de données qu'on lui soumet, grâce à un jeu de macros internes complexes, tout en restant simple d'utilisation. Ceci offre plusieurs avantages : la garantie d'un surcoût nul en mémoire et en performances par rapport à une bibliothèque encapsulant dynamiquement les structures de données, la génération de code machine minimale identique à celui d'une implémentation directe, un code source qui reste facile à relire, et l'interchangeabilité des algorithmes conteneurs.

La figure 4.7 illustre l'utilisation de `gpct` mettant en œuvre une liste simplement chaînée de la structure C nommée `myitem`. Ici le conteneur est protégé par un verrou de la bibliothèque de *thread*s *POSIX*.

```
/* user data structure */
struct myitem
{
  CONTAINER_ENTRY_TYPE(SLIST)   list_entry;
  const char                    *data;
};

/* use pthread mutex when accessing linked list */
#define CONTAINER_LOCK_mylist PTHREAD_MUTEX

/* define linked list type and access functions */
CONTAINER_TYPE       (mylist, SLIST, struct myitem, list_entry);
CONTAINER_FUNC       (mylist, SLIST, static, myfunc);

void myfunc()
{
  /* declare list container and item object */
  mylist_root_t list;
  struct myitem a = { .data = "test" };

  myfunc_init(&list);
  /* add and remove object from list */
  myfunc_push(&list, &a);
  myfunc_pop(&list);
}
```

FIGURE 4.7 – Exemple de liste chaînée simple avec verrouillage par mutex

Cette bibliothèque est adaptée au développement dans les environnements parallèles et multiproces-

seurs qu'on souhaite mettre en place pour le support des plateformes hétérogènes, puisqu'elle offre la possibilité de protéger les structures de données par des accès atomiques dont les mécanismes ne sont pas imposés.

4.2.2 La bibliothèque C

MutekH fournit au travers du module (`libc`) un sous-ensemble de la bibliothèque C *POSIX*, dans la mesure de ce qui est envisageable dans un environnement noyau :

– Des fonctions courantes standards permettant de manipuler les chaînes de caractères et de copier des blocs mémoire, qui sont utilisées dans tout le reste du noyau.
– Des services de formatage de texte, d'affichage et de gestion de flux.
– Des services d'accès aux fichiers.
– Des services d'allocation mémoire.
– Des primitives de synchronisation utiles dans les environnements multiprocesseurs comme les sémaphores *POSIX*.

D'une façon générale, la bibliothèque C *POSIX* est particulière car elle propose une palette de services très différents :

– certains sont absolument indispensables et même parfois liés de manière intrinsèque au code généré par le compilateur,
– d'autres sont utiles uniquement pour que les applications existantes puissent s'exécuter en mode noyau sans réécriture. C'est le cas, par exemple, de `open`, `read`, `write`, `close`, désignant des flux via des numéros de descripteurs entiers, originellement conçus ainsi en raison de l'impossibilité d'échanger des pointeurs au travers des appels système *UNIX*.
– certains appels n'ont simplement pas de sens dans un noyau, c'est le cas par exemple des fonctions de gestion des processus et signaux *UNIX*, telles que `exit`, `fork` ou encore `kill`. .

Les fonctions qui sont utiles seulement pour le portage d'applications, et qui ne sont donc pas utilisées dans le reste du code du noyau, doivent être activées explicitement dans la configuration. La bibliothèque C de *MutekH* se rapproche des bibliothèques C fournies par les autres systèmes supportant les applications qui s'exécutent en mode noyau. C'est le cas de *RTEMS* ou *eCos*. Ces bibliothèques sont plus étendues que les bibliothèques C de noyaux comme *Linux* qui ne proposent pas certaines fonctions. Il ne faut pas confondre ces bibliothèques avec celles proposées en mode utilisateur sous *UNIX*, environnement pour lequel des standards comme *POSIX* décrivent l'implémentation complète pour ce contexte particulier.

4.2.3 La couche d'abstraction Hexo

La couche d'abstraction du matériel fournit les services essentiels à l'hétérogénéité. Ces services permettent de garantir l'uniformité du comportement des différents types de processeurs vis-à-vis de la plateforme. Ces mécanismes ont déjà été présentés dans la section 3.4.

L'implémentation de cette couche d'abstraction a été réalisée de manière à minimiser et même supprimer le coût de l'abstraction. En effet, la modularité du logiciel n'exclut pas la définition de nombreuses fonctions `static inline` dans les fichiers entêtes. Il s'agit ici de modularité statique, et

non de modularité dynamique qu'on traiterait par pointeurs sur fonctions : tout se trouve donc réduit à la compilation. Même si le code source est localisé et regroupé par type de processeur et par type de plateforme, la configuration sélectionne les fichiers à inclure et le compilateur se charge d'*inliner* le code.

Hexo reste donc une couche très fine qui, une fois traitée par le compilateur, est alors complètement intégrée dans les couches plus hautes du noyau. Les services présentés précédemment sont fournis au travers de divers entêtes :

- `hexo/init.h` : démarrage et initialisation du système
- `hexo/cpu.h` : invalidation du cache et accès aux registres spécifiques du processeur
- `hexo/atomic.h` : services d'accès atomiques à la mémoire
- `hexo/ordering.h` : sérialisation des accès à la mémoire par le compilateur et le processeur
- `hexo/lock.h` : service de verrous de type *spinlock*
- `hexo/endian.h` : services d'accès mémoire non-alignés et d'endianness non native
- `hexo/iospace.h` : services d'accès à l'espace d'adressage des périphériques
- `hexo/local.h` : variables contextuelles.
- `hexo/mmu.h` : support de la mémoire virtuelle.
- `hexo/types.h` : définitions de types entiers portables
- `hexo/context.h` : gestion des contextes d'exécution
- `hexo/interrupt.h` : gestion des interruptions, exceptions et appels système
- `hexo/ipi.h` : gestion des interruptions inter-processeurs

Outre la modularité et l'attention particulière apportée lors de la réalisation, afin que ces différents services se comportent de la même manière avec des processeurs différents, l'implémentation et l'API de la plupart de ces services restent classiques. Le support de l'hétérogénéité est assuré davantage par le choix de l'API et sa généricité que par son implémentation, ainsi que nous l'avons exposé dans les solutions. Seuls quelques services, telle la gestion des contextes d'exécution, méritent une présentation plus détaillée.

Gestion des contextes d'exécution

Hexo propose une *API* générique pour sauvegarder et restaurer un contexte d'exécution indépendamment du type de processeur. La couche d'abstraction du matériel offre uniquement le service de sauvegarde, c'est-à-dire le stockage de l'état du processeur dans un contexte dédié en mémoire. Ce contexte contient les valeurs des registres du processeur et reste spécifique au type de processeur.

La figure 4.8 donne un exemple d'utilisation de l'API. Dans cet exemple deux contextes d'exécution peuvent être sauvegardés dans les variables `ctx_a` et `ctx_b` et utilisent les piles déclarées par `stack_a` et `stack_b`. Chaque contexte invoque sa sauvegarde et la restauration de l'autre contexte dans une boucle sans fin.

Il est important de noter que dans un système hétérogène, un contexte sauvegardé par un processeur, ne peut être restauré que sur un processeur du même type. Cette limitation est en accord avec le type d'hétérogénéité visée et assure la correspondance du code, avec une granularité au niveau fonctions, entre les différents processeurs. La sauvegarde du contexte d'exécution pouvant s'effectuer à n'im-

```
static CONTEXT_ENTRY(context_entry)
{
  struct context_s *other = param;

  while (1)
    context_switch_to(other);
}

/* execution contexts and associated stacks */
struct context_s ctx_a, ctx_b;
uint8_t          stack_a[512], stack_b[512];

void main()
{
  /* initialize context A and pass context B as entry parameter */
  context_init(&ctx_a, stack_a, stack_a + 512, context_entry, &ctx_b);

  /* initialize context B and pass context A as entry parameter */
  context_init(&ctx_b, stack_b, stack_b + 512, context_entry, &ctx_a);

  /* discard current context and restore context A */
  context_jump_to(&ctx_a);
}
```

FIGURE 4.8 – Exemple de permutations infinies entre deux contextes A et B

porte quel instant de l'exécution du contexte dans un système multitâche préemptif, l'interruption d'un contexte à l'intérieur d'une fonction ne permet pas de la restaurer sur un autre processeur.

Un changement de contexte sur appel de fonction n'est pas viable pour autant : un autre obstacle est l'utilisation de conventions d'appels (*ABI*) différentes entre les processeurs. Ceci implique que le format de la pile d'exécution d'un contexte diffère selon le processeur sur lequel il s'exécute. Il n'est donc pas envisageable de réaliser la migration d'un contexte entre deux types de processeurs sans définir une *ABI* spécifique commune, car ceci impliquerait de réécrire une partie des outils de compilation.

Cette contrainte de migration de contexte sur un même type de processeur n'est pas une limitation forte puisque l'hétérogénéité est généralement mise à profit pour exploiter les atouts des différents processeurs et leur faire exécuter des tâches spécifiques.

Plateformes et processeurs supportés

Rappelons qu'Hexo distingue le code spécifique à la plateforme du code spécifique au processeur. Les plateformes supportées sont les suivantes :

- `simple` : Architecture monoprocesseur simple correspondant à des plateformes à base de micro-contrôleurs.
- `ibmpc` : Support de l'architecture des serveurs et stations de travail multiprocesseurs classiques de type PC.
- `soclib` : Support des plateformes *SoCLib* multiprocesseurs.
- `emu` : Support de l'exécution du noyau et de l'application dans un processus *UNIX*.

Les différents types de processeurs supportés par Hexo à ce jour sont les suivants :

– Les processeurs x86 32 bits et x86_64.
– Les processeurs PowerPC
– Les processeurs Mips 32bits big endian et little endian.
– Les processeurs ARM
– Les microcontrôleurs Atmel AVR.

L'architecture utilisée pour les expérimentations sur des plateformes multiprocesseurs hétérogènes est l'architecture *SoCLib*.

Le support de l'architecture `ibmpc` a permis d'assurer un degré important de généricité du code tant cette architecture diffère d'une architecture moderne employant des processeurs *RISC*. Ceci a également permis de tester le noyau sur une plateforme multiprocesseur éprouvée et totalement fiable.

L'architecture `emu` est particulière puisqu'elle permet l'exécution du noyau *MutekH* dans un processus utilisateur au-dessus du noyau *Linux* ou *Darwin*. Son principe est similaire à celui de *User-mode Linux* [13]. Dans ce mode, le binaire du noyau est une application indépendante, liée à aucune bibliothèque du système d'exploitation hôte. Seuls quelques appels système directs sont effectués vers le noyau : pour allouer un unique bloc mémoire servant de RAM, ou encore pour accéder aux entrées/sorties. `emu` supporte les architectures multiprocesseurs en créant plusieurs processus et en employant les services de mémoire partagée du noyau hôte. Comme sur les architectures `soclib` et `ibmpc`, il est possible d'obtenir de la vraie concurrence si la machine hôte est munie de plusieurs processeurs. Cette architecture a permis de tester tous les aspects algorithmiques du noyau sans recourir à du matériel spécifique, avec une vitesse d'exécution native, tout en facilitant l'utilisation d'outils de débogage.

4.2.4 Les services de l'exo-noyau indépendants du matériel

Le module `mutek` regroupe tous les services de l'exo-noyau qui sont indépendants du matériel. Une des fonctions de l'exo-noyau est le partage et le multiplexage des ressources pour les services de plus haut niveau utilisés par les applications. Ces services de plus haut niveau sont les bibliothèques proposant les diverses interfaces standardisées.

L'exo-noyau de MutekH fournit les services suivants :

– Ordonnanceur de contextes d'exécution.
– Primitives de synchronisation basées sur l'ordonnanceur.
– Allocation de la mémoire.
– Base de temps globale abstraite du matériel.
– Centralisation des messages vers la console.

Ordonnancement dans les systèmes hétérogènes

L'ordonnanceur utilise la sauvegarde/restauration de contextes d'exécution, fournie par le module `hexo`. Ainsi que nous l'avons souligné, un contexte sauvegardé ne peut être restauré que sur un

processeur du même type. Ceci interdit la migration d'un contexte déjà démarré sur un processeur de type différent.

Cependant l'ordonnancement des contextes doit être générique et global. Ceci est nécessaire pour qu'une tâche s'exécutant sur un certain processeur puisse interagir avec des tâches s'exécutant sur un processeur d'un autre type, et ce, en utilisant les primitives de synchronisation standards. Ainsi l'ordonnanceur fourni par le module `mutek` définit plusieurs files d'exécution selon la politique d'ordonnancement choisie et permet de définir et manipuler des files d'attente pour l'implémentation de toutes sortes de primitives de synchronisation.

Il est fréquent qu'un processeur d'un certain type ait à modifier la file d'exécution des contextes associée à un processeur d'un autre type. Ceci est un exemple de mécanisme du noyau qui s'appuie fortement sur le partage des variables en mémoire et sur les accès atomiques entre processeurs de différents types.

La figure 4.9 présente un exemple de primitive de synchronisation basée sur l'ordonnanceur de *MutekH*. Cette primitive de synchronisation de type `pthread_mutex_t` est un verrou. Elle est réalisée à l'aide d'une file d'attente de l'ordonnanceur et d'un booléen indiquant la prise du verrou. La fonction `pthread_mutex_lock` permet de prendre le verrou et place le contexte appelant en attente, seulement si le verrou est déjà pris. La fonction `pthread_mutex_unlock` tente de réveiller le contexte suivant, en attente sur le verrou, et marque le verrou comme libre le cas échéant.

On note que tous les mécanismes permettant le support natif de l'hétérogénéité sont exploitables par les bibliothèques standards et par les applications, mais qu'ils permettent également l'utilisation d'algorithmes et d'approches classiques dès les couches basses du noyau.

Spécialisation des contextes d'exécution

L'approche exo-noyau implique que l'ordonnanceur de contextes reste générique et propose des contextes non spécialisés. Ces contextes génériques permettent d'implémenter des types de contextes variés, tels les *thread*s *POSIX* ou les processus *UNIX*.

Le service de variables contextuelles fourni par *Hexo* se révèle très utile dans cette situation car il permet à chaque bibliothèque de haut niveau de spécialiser un contexte d'exécution en lui accrochant des données qui lui sont propres. Ainsi chaque bibliothèque de parallélisation peut proposer des contextes d'exécution plus complexes que les contextes de base fournis par *Hexo*, en les spécialisant via des attributs supplémentaires.

De la même manière, *Hexo* permet de prendre en charge des routines de traitement des exceptions et des routines de traitement des appels système utilisateurs, propres à un contexte d'exécution.

Une bibliothèque implémentant *UNIX* peut par exemple accrocher toutes les données propres à un processus *UNIX*, telles que les descripteurs de fichiers ouverts, dans des variables globales locales au contexte. Il est aussi possible de définir une routine de traitement des exceptions propre à ce contexte, liée à la gestion des signaux *UNIX*. D'autres contextes s'exécutant simultanément peuvent être spécialisés différemment par une autre bibliothèque de parallélisation.

```
typedef struct      pthread_mutex_s
{
  bool_t             locked;      /* currently locked */
  sched_queue_root_t wait;        /* wait queue */
}                    pthread_mutex_t;

error_t pthread_mutex_lock(pthread_mutex_t *mutex)
{
  sched_queue_wrlock(&mutex->wait);

  if (mutex->locked) {
      /* already locked, wait on queue */
      sched_wait_unlock(&mutex->wait);
  } else {
      /* mark mutex as locked */
      mutex->locked = 1;
      sched_queue_unlock(&mutex->wait);
  }

  return 0;
}

error_t pthread_mutex_unlock(pthread_mutex_t *mutex)
{
  sched_queue_wrlock(&mutex->wait);

  /* try to wake any waiting thread */
  if (!sched_wake(&mutex->wait))
    mutex->locked = 0;

  sched_queue_unlock(&mutex->wait);
  return 0;
}
```

FIGURE 4.9 – Exemple d'implémentation de mutex utilisant les primitives de l'ordonnanceur

Primitives de synchronisation

Rappelons que l'ordonnanceur permet l'implémentation de primitives de synchronisation standards. Certains services du noyau implémentés dans des bibliothèques de services, comme la gestion du système de fichiers, la couche réseau ou les accès bloquants aux périphériques, ont besoin de primitives de synchronisation.

L'utilisation de primitives de synchronisation de haut niveau, à titre d'exemples celles qui sont définies dans la bibliothèque de *thread*s *POSIX*, n'est pas souhaitable dans le noyau pour des raisons de dépendances et de modularité. En effet, leur utilisation poserait deux problèmes : elle forcerait l'activation de la bibliothèque des *thread*s *POSIX* au niveau de la configuration, alors que de telles primitives sont normalement réservées à l'application ; et elle permettrait l'utilisation des services du noyau exclusivement depuis des contextes d'exécution de type *thread*s *POSIX*. On ne peut imposer à l'application l'utilisation des primitives de synchronisation *POSIX* à la seule activation de la couche réseau ou le système de fichiers, par exemple.

Le module `mutek` fournit donc un jeu de primitives de synchronisation génériques telles que des

sémaphores et des verrous de type *rwlock* permettant de distinguer les accès en lecture et en écriture. Celles-ci peuvent être invoquées depuis n'importe quel contexte d'exécution.

Allocation de la mémoire

Le module `mutek` propose différents algorithmes d'allocation de complexités diverses. Tout comme l'ordonnanceur de contextes, l'allocateur mémoire travaille sur des structures de données avec accès concurrents entre processeurs.

L'allocateur mémoire basique utilisé initialement par *MutekH* a rapidement été remplacé par une version plus évoluée permettant le placement des données pour le support des architectures *NUMA*. Cette évolution a été effectuée grâce à des projets dépassant le cadre de cette thèse et sans rapport avec l'hétérogénéité. Il est intéressant de noter que la couche d'abstraction *Hexo* a permis aux nouveaux algorithmes implémentés, de fonctionner directement sur des plateformes hétérogènes, sans effort ni attention particulière de la part de leurs développeurs.

4.3 Les services standards de parallélisation

L'architecture en exo-noyau permet le développement de bibliothèques standards au-dessus des couches basses de l'exo-noyau, constituées par les modules `hexo` et `mutek`. Il s'agit ici de proposer des services relatifs à l'exécution parallèle et à la gestion des tâches et contextes d'exécution, services indispensables dans un contexte multiprocesseur.

Une des motivations principales du support natif de l'hétérogénéité par le système d'exploitation est de rendre accessible les plateformes hétérogènes aux applications déjà existantes, alors que les autres approches impliquent un développement spécifique. Le support de bibliothèques standards telles que *PThread* ou *OpenMP* permet à la fois d'assurer la généricité de la solution, de valider l'implémentation et d'élargir le spectre d'utilisation du noyau.

Cette section présente les principales bibliothèques de parallélisation proposées par *MutekH*.

4.3.1 La bibliothèque de Threads POSIX

Cette bibliothèque incontournable dans le domaine de la programmation parallèle est proposée par le module `libpthread`, elle est un sous-ensemble de l'API *POSIX*. La norme *POSIX* spécifiant l'ensemble de l'API d'un système de type *UNIX*, elle est liée à la notion de processus et aux concepts propres à *UNIX* d'une manière générale. Ceci soulève des problèmes similaires à ceux que l'on a déjà mentionnés à propos de la bibliothèque C standard, quand on souhaite une implémentation sortant du cadre d'*UNIX*.

L'extension *Threads extensions* de la norme *POSIX* reste suffisamment indépendante des concepts d'*UNIX* pour permettre l'implémentation d'un large sous-ensemble de services indépendants d'*UNIX*. Dans cette bibliothèque, le contexte d'exécution est un *thread* et sa spécification reste abstraite de la notion de processus *UNIX*. Seules quelques fonctions relatives à la gestion des signaux et à la mémoire partagée des processus *UNIX* doivent être laissées de côté. Cette approche est retenue par de nombreux noyaux pour systèmes embarqués proposant également cette bibliothèque.

Les primitives de synchronisation

Deux types de services sont proposés par cette bibliothèque : la possibilité de créer des nouveaux contextes d'exécution et la possibilité de synchroniser leur exécution au travers de diverses primitives. La figure 4.9 montre comment les couches basses du noyau *MutekH* permettent d'implémenter la primitive de synchronisation *mutex* de *POSIX*.

Des versions étendues de l'API proposent des primitives de synchronisation supplémentaires : la primitive *rwlock* à exclusion mutuelle avec distinction entre les lecteurs et écrivains, ou encore la primitive *spinlock* de verrous à attente active.

Le système de configuration de *MutekH* permet de choisir de manière fine quelles primitives de synchronisation doivent être supportées par le noyau.

La placement des tâches

Le placement des threads sur un processeur particulier, lors de leur création, n'est pas pris en charge par l'API standard de la bibliothèque *POSIX*, cependant des extensions permettent de forcer l'assignation d'un thread sur un processeur. Cette fonctionnalité est importante dans le cadre d'un système hétérogène où l'exécution d'une tâche particulière peut être adaptée à un type de processeur.

L'ordonnanceur de *MutekH* définit des files d'exécution pour un processeur ou pour un groupe de processeurs, selon la politique de migration de contextes choisie dans la configuration. La bibliothèque `libpthread` propose un attribut non standard utilisable avec la fonction `pthread_create`. Cet attribut permet à la bibliothèque de choisir les files d'exécution adaptées. La figure 4.10 donne un exemple de code associant un thread à un processeur.

Dans le cadre d'un système hétérogène, lorsque l'ordonnanceur est configuré pour disposer d'une file d'exécution par processeur, cet attribut force l'exécution du thread sur le processeur spécifié. Quand aucun attribut n'est utilisé le thread s'exécute sur le processeur qui exécute la fonction de création.

Lorsque l'ordonnanceur est configuré pour partager une file d'exécution entre plusieurs processeurs, il existe au moins une file d'exécution par type de processeur et le thread n'est élu que par les processeurs désignés par l'attribut. Si aucun attribut n'est utilisé, le thread est libre de s'exécuter sur n'importe quel processeur du même type lors de chaque élection.

```
pthread_t th;
pthread_attr_t at;

pthread_attr_init(&at);

/* assign thread to cpu 0 */
pthread_attr_affinity(&at, 0);

pthread_create(&th, &at, th_func, NULL);

pthread_attr_destroy(&at);
```

FIGURE 4.10 – Exemple d'affectation d'un thread à un processeur lors de sa création

4.3.2 La bibliothèque OpenMP

Le standard *OpenMP* propose une extension du langage de programmation C (mais aussi C++ et Fortran), permettant d'exécuter certaines sections de code en parallèle dans différents contextes d'exécution.

La création de *threads* n'est pas explicite dans le code de l'application qui se contente d'exprimer quelles parties du programme peuvent être exécutées en parallèle via des directives de compilation dédiées. La figure 4.11 donne un exemple simple d'utilisation d'*OpenMP*.

```
size_t i;
char data[100];

#pragma omp parallel for
for (i = 0; i < 100; i++)
  data[i] *= 2;
```

FIGURE 4.11 – Exemple de code C utilisant une directive *OpenMP* de parallélisation de boucle

L'implémentation d'*OpenMP* nécessite la collaboration du compilateur et d'une bibliothèque associée. Cette bibliothèque prend en charge la création de contextes d'exécution et leur synchronisation. Les fonctions de création d'un contexte d'exécution doivent respecter les attentes du code généré par le compilateur pour réaliser ces actions, bien que les appels à ces fonctions ne soient pas explicites dans le code source.

Le module `libgomp` de *MutekH* est un portage de la bibliothèque *OpenMP* associée au compilateur C *GNU*. Cette bibliothèque a été conçue de manière portable et dispose de plusieurs modules pour supporter les environnements d'exécution parallèles de différents systèmes d'exploitation. L'ajout du support pour *MutekH* en a été facilité.

Le standard *OpenMP* ne prévoit pas de spécifier quels processeurs doivent être utilisés pour l'exécution parallèle d'un bloc de code donné. Cependant plusieurs politiques existent quant à l'allocation et la création des contextes d'exécution qui seront utilisés par la suite pour traiter le parallélisme.

L'utilisation d'*OpenMP* standard dans le cadre des systèmes hétérogènes permet donc de paralléliser l'application, mais elle ne permet pas de choisir dynamiquement le type de processeur à utiliser pour un bloc de code donné. Ceci est une limitation empêchant qu'une application basée sur *OpenMP* tire parti de l'hétérogénéité de la plateforme, mais n'exclut pas sa cohabitation avec d'autres bibliothèques de parallélisation.

OpenMP propose une directive permettant l'exécution d'un bloc de code donné, sur le processeur maître uniquement. Dans la même optique de sélection des processeurs, il serait tout à fait envisageable d'étendre *OpenMP* pour sélectionner le type de processeur devant traiter un bloc de code dans une plateforme hétérogène. Cette approche sort du cadre de cette thèse car elle nécessite la modification du compilateur et ne traite pas le problème des applications existantes.

4.3.3 Le projet CAPSULE

Certaines bibliothèques dédiées à la parallélisation d'applications ont été spécifiées ou simplement développées au-dessus de l'exo-noyau de *MutekH*.

On peut citer l'API Capsule [27] spécifié par l'Inria et dont une implémentation spécifique annexe a été réalisée dans le module `libcapsule` du projet *MutekH*. Cette bibliothèque permet de définir des tâches dont le degré de parallélisation est dynamiquement adapté selon la charge du système.

4.3.4 Utilisation dans SoCLib

La plateforme de prototypage virtuel *SoCLib* propose, entre autres, le système d'exploitation *MutekH* pour supporter un vaste ensemble de plateformes monoprocesseurs et multiprocesseurs.

On peut citer l'outil d'exploration architecturale *DSX*. Cet outil permet de générer le code de la plateforme matérielle conjointement au code logiciel de l'application. L'ensemble de l'application ainsi générée est basée sur un graphe de tâches communiquantes logicielles ou matérielles de type *KPN*[19].

Dans ce cadre, *MutekH* fournit la bibliothèque permettant de mettre en place statiquement le graphe de tâches communiquantes et les *FIFO*, via le module `libsrl` [16]. Cette bibliothèque fournit les primitives de synchronisation et les primitives d'accès aux *FIFO* qui peuvent être partagées par des tâches matérielles.

4.3.5 Parallélisation en mode utilisateur

Les différentes bibliothèques permettant la parallélisation, présentées jusqu'ici, servent les applications qui s'exécutent en mode noyau, c'est-à-dire avec le même niveau de privilège que le noyau. Cette approche est généralement adaptée au domaine de l'embarqué.

La couche d'abstraction du matériel *Hexo* sait gérer la séparation des privilèges et le mode utilisateur des différents processeurs qu'elle supporte. En effet, l'exo-noyau de MutekH a été conçu pour que des bibliothèques de haut niveau puissent proposer différentes interfaces de systèmes d'exploitation standards. Le développement encore partiel d'un module `libunix`, visant à développer une implémentation d'*UNIX*, utilise le support du mode utilisateur et des appels système fourni par *Hexo*.

4.4 Les autres services

Le portage d'applications existantes sur une plateforme hétérogène est rendu possible grâce au support natif de l'hétérogénéité et aux services de parallélisation présentés jusqu'ici. Ces applications dépendent toutefois d'autres services essentiels à leur fonctionnement.

Cette section présente très brièvement les services disponibles dans *MutekH*.

4.4.1 Les pilotes de périphériques

Différents pilotes de périphériques sont disponibles et supportent le matériel couramment utilisé dans les plateformes prises en charge par *Hexo*.

La prise en charge des périphériques sous *UNIX* établit deux classes de périphériques, courantes à l'époque de son développement initial : Les périphériques de blocs et les périphériques de caractères. Tous les autres types de périphériques gérés par les implémentations modernes de systèmes *UNIX* sont pris en charge au travers de l'appel système `ioctl` qui a été conçu pour traiter les cas spécifiques.

Fort de ce constat, les pilotes de périphériques dans *MutekH* se basent sur un nombre de classes de périphériques plus important et extensible. Il existe une API différente pour chaque classe de périphériques supportés :

- `block` : Gestion des périphériques de stockage de masse.
- `char` : Gestion des périphériques d'entrée/sortie de caractères.
- `icu` : Gestion et cascade des contrôleurs d'interruptions.
- `timer` : Gestion des périphériques de base de temps.
- `enum` : Gestion des énumérateurs de bus.
- `fb` : Gestion des périphériques d'affichage graphique.
- `gpio` : Gestion des lignes d'entrée/sortie logiques, d'usage général, courantes dans les systèmes embarqués.
- `i2c` : Gestion des périphériques sur bus *I2c*.
- `input` : Gestion des périphériques de saisie à états.
- `net` : Gestion des interfaces réseau.
- `sound` : Gestion des périphériques de son.

Comme c'est le cas dans certains noyaux, les périphériques peuvent être virtuels : un périphérique de *ramdisk* ou de cache disque par exemple.

4.4.2 Les services propres au noyau

L'architecture en exo-noyau implique le multiplexage des ressources et leur utilisation par différentes implémentations d'interfaces de systèmes d'exploitation, parfois simultanément dans un même noyau en exécution. Certains services habituellement fournis par le système d'exploitation doivent donc être mutualisés, comme c'est le cas pour l'ordonnanceur du module `mutek`. *MutekH* incorpore donc différents services couramment fournis par les noyaux de systèmes d'exploitation, factorisés dans des modules dédiés.

Ces bibliothèques sont destinées à la fois aux applications en mode noyau et aux bibliothèques implémentant une interface standard de plus haut niveau :

- Le module `libvfs` fournit toute la gestion du système de fichiers dans le noyau. Elle permet de gérer plusieurs arborescences disjointes, et possède la notion de liens, de références multiples aux fichiers et de points de montage. Différents pilotes de systèmes de fichiers associés permettent l'accès aux périphériques de blocs ainsi qu'à des systèmes de fichiers virtuels.

– Le module `libnetwork` propose une implémentation de couche réseau TCP/IP avec les protocoles de bas niveau classiques ARP, IP, ICMP, UDP, TCP et un support DHCP et NFS.
– Le module `libelf` permet le chargement en mémoire et la manipulation de fichiers binaires *ELF*.
– Le module `libcrypto` propose des primitives cryptographiques diverses.
– Le module `libfdt` propose un parseur de *Flattened Device Tree*s [12], format de description du matériel passé en paramètre au noyau par divers chargeurs de démarrage.

Les API de ces bibliothèques sont propres au noyau, cependant certains de leurs services sont accessibles également au travers de la bibliothèque C standard.

4.4.3 Les bibliothèques utilitaires

Quelques bibliothèques facilitent le développement d'applications en mode noyau et ne sont utiles que dans ce cas. *MutekH* propose les bibliothèques utilitaires suivantes :

– Ainsi qu'on l'a expliqué précédemment, le module `libc` fournit un ensemble de fonctions qui ne sont utiles qu'au portage d'applications en mode noyau.
– Le module `libm` propose une bibliothèque mathématique standard.
– Le module `liblua` fournit un interpréteur du langage de script *Lua*, particulièrement léger et adapté à l'embarqué.
– Le module `libtermui` propose des fonctionnalités similaires à la bibliothèque *GNU readline*, permettant la mise en place d'interfaces utilisateur en mode console, avec complétion et historique des commandes.

4.5 Conclusion sur l'implémentation du noyau

La modularité extrême de la solution mise en place, imposée par l'architecture en exo-noyau, se révèle adaptée au support de l'hétérogénéité. En effet, elle permet de cloisonner le code spécifique à la gestion de l'hétérogénéité dans les couches basses et d'exploiter ces mécanismes pour l'implémentation d'algorithmes classiques pour tous les services de base de l'exo-noyau.

Cette approche permet également d'apporter le service de l'hétérogénéité native à une large palette de bibliothèques standards de parallélisation et de toucher un maximum d'applications existantes. Outre l'élargissement du terrain d'expérimentation, ceci assure qu'on propose une solution adaptée à la réalité des différentes utilisations qui sont habituellement faites d'un noyau, mais aussi d'en apercevoir les limites lors de l'utilisation sur une plateforme hétérogène.

Le noyau mis en place ici permet une réelle transparence de l'hétérogénéité de la plateforme tout en respectant les standards d'*API* de bibliothèques répandues et les *ABI* des processeurs.

Ce développement a permis de relever le défi de la modularité et de la portabilité et répond tout à fait à la problématique initiale du support natif de l'hétérogénéité et de la prise en charge des applications existantes. En plus des services de base indispensables à l'évaluation de l'hétérogénéité, les choix retenus ont permis le développement d'un noyau complet utilisable et utilisé dans d'autres projets.

Chapitre 5

Edition de liens pour architectures hétérogènes

Un des aspects importants du support natif de l'hétérogénéité est le partage de l'intégralité des données en mémoire entre les différents processeurs. Les différents algorithmes du noyau et de l'application, qui partagent en effet toutes les variables et données allouées, doivent pouvoir y accéder de manière transparente comme dans le cas d'un système multiprocesseur à mémoire partagée classique. De plus les fonctions possédant un code source identique doivent être situées à des adresses identiques pour les raisons déjà évoquées dans le chapitre 3.

La compilation et la construction des binaires déterminent l'ordre d'apparition des différents symboles - fonctions et variables - dans les sections du fichier objet et par conséquent, dans la mémoire.

La solution proposée dans la section 3.5.3 implique une étape d'uniformisation et de réorganisation des symboles, habituellement classés dans un ordre aléatoire, à l'intérieur des fichiers objets. En effet, une modification en profondeur des fichiers objets est nécessaire après l'étape de compilation. Ce chapitre détaille ces modifications et présente les outils qui ont été développés pour atteindre ce but.

Les notions relatives aux symboles, aux relocations et plus généralement à la structure des fichiers objets, déjà abordées dans la section 3.5.3, sont nécessaires à la compréhension de ce chapitre [20].

5.1 Manipulation du format elf

Le format de fichiers *ELF* (*Executable and Linkable file Format*) est largement adopté dans le monde d'*UNIX* et des logiciels libres. C'est également le format de fichier objet et exécutable le plus moderne et le plus flexible. C'est donc ce format qui est utilisé pour la génération de binaires hétérogènes avec *MutekH*.

Il définit plusieurs types de contenus stockés dans les sections qui composent le fichier. Certaines sections contiennent le code et les données qui sont chargées en mémoire :

– La section `.text` contient le code.

– La section `.rodata` contient les données en lecture seule.
– La section `.data` contient les variables globales non affublées de l'attribut `const`.
– La section `.bss` contient les variables globales non initialisées ou nulles.

Certaines sections ne sont pas nécessairement chargées en mémoire et contiennent des méta-informations relatives au fichier objet ou exécutable, on peut citer :

– Une table des symboles qui permet de nommer chaque fonction ou variable globale. Les entrées de cette table associent des noms de symboles à leur adresse et à leur taille dans la section où elles résident.
– Des tables de relocations. Les entrées de ces tables permettent de retoucher le contenu de certaines sections pour prendre en compte l'adresse d'un symbole non résolu jusqu'ici ou l'adresse de chargement de la section encore inconnue dans le fichier objet. Ainsi le code binaire d'une instruction ou d'une donnée peut être modifié ultérieurement au processus d'assemblage. Une entrée de cette table associe le symbole dont l'adresse est encore inconnue à l'adresse qui doit être retouchée dans la section. Un fichier exécutable non dynamique ne possède plus de relocations.

Une bibliothèque C++ permettant de charger en mémoire une représentation du fichier ELF a été développée dans le cadre de cette thèse. Elle permet de désérialiser les fichiers ELF pour manipuler chaque élément comme un objet C++. De cette manière chaque symbole, chaque relocation ainsi que les relations entre les divers éléments, peuvent être modifiés avant d'être sérialisés vers un nouveau fichier.

5.1.1 Hiérarchisation du format elf

Le fichier ELF contient l'ensemble des éléments stockés à plat :

– Les variables et fonctions sont stockées de façon contigüe dans les sections.
– Les différentes entrées de symboles sont stockées dans une unique table.
– Les relocations sont stockées dans une table pour chaque section qu'elles retouchent, c'est-à-dire généralement pour chaque section chargée en mémoire.

Cette organisation à plat (figure 5.1) convient quand on ne souhaite pas modifier l'ordre des symboles dans une section. En effet tout est prévu afin de pouvoir reloger une section de données ou de code d'un seul bloc, l'ordre des éléments dans la section étant figé une fois pour toutes à l'assemblage. Cette contrainte doit être supprimée pour résoudre le problème d'édition des liens hétérogène.

Pour cette raison, une représentation hiérarchique est plus appropriée et permet d'entreprendre les modifications nécessaires avec une flexibilité maximale. Le processus de désérialisation organise les objets C++ correspondant aux éléments du fichier *ELF* comme suit :

– Une liste de symboles est établie pour chaque section,
– le contenu des variables et des fonctions est accroché à chaque symbole,
– une liste de relocations est établie pour chaque symbole qu'elles désignent.

Le contenu de chaque symbole n'est donc plus stocké dans la section, mais accroché au symbole correspondant. Ceci permet d'établir un nouvel ordre de stockage des symboles pour une section, ainsi les différentes fonctions ou variables seront réécrites dans l'ordre choisi lors de la sérialisation à venir.

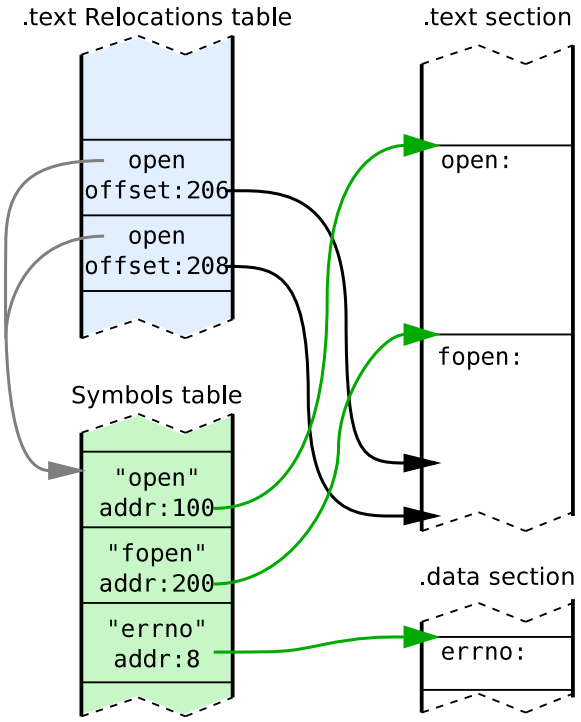

FIGURE 5.1 – Eléments stockés dans les tables du fichier ELF

La figure 5.1 montre les tables de symboles et de relocations indexant les éléments présents dans les sections `.text` et `.data`, telles qu'elles sont représentées dans le fichier objet. On observe ici, par exemple, que l'adresse de la fonction `open` reste à remplacer en deux endroits de la fonction `fopen`.

5.1.2 Calcul des adresses relatives

L'organisation hiérarchique seule n'est pas suffisante. L'organisation à plat du fichier ELF permet plusieurs simplifications courantes, par les compilateurs et assembleurs, qui compliquent particulièrement le travail de réorganisation.

Etant donné qu'une variable ou un symbole garde habituellement une position invariante dans la section de données ou de code à l'intérieur d'un même fichier objet, les adresses des entrées de relocation sont relatives au début de la section. C'est-à-dire qu'une entrée de relocation ne désigne pas explicitement le symbole qui sera modifié, mais simplement l'adresse de la modification dans la section.

Quand une fonction est déplacée, par exemple, les relocations qui vont modifier certaines instructions lors de l'édition des liens finale doivent être ajustées également si la fonction a été déplacée.

Afin de résoudre ce problème, une passe supplémentaire est nécessaire pour rendre les relocations relatives aux symboles qu'elles modifient :

– Recherche du symbole modifié par une relocation à partir de l'adresse
– Calcul de l'adresse de la relocation relative au symbole.

– Ajout d'une relation entre la relocation et le symbole modifié.

Ainsi, dans le fichier, la relocation associe un symbole dont l'adresse reste inconnue, à l'adresse dans la section où il faudra écrire le valeur du symbole (fig. 5.1), tandis que dans la représentation utilisée, une relocation associe un symbole dont l'adresse reste inconnue, au symbole qu'il faudra modifier (fig. 5.2)

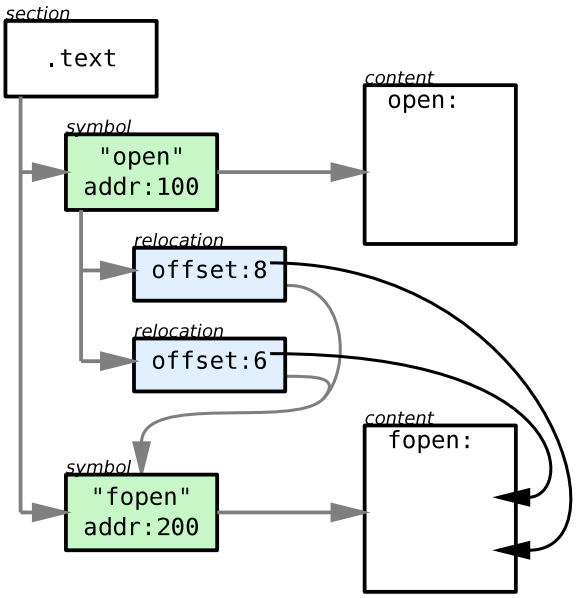

FIGURE 5.2 – Représentation désérialisée de la section `.text` du fichier ELF présenté figure 5.1

5.1.3 Optimisations du compilateur

Certaines optimisations du compilateur se basent sur le fait que la position relative entre deux fonctions ou deux variables d'un même fichier source est connue à la compilation, bien que cette position relative soit choisie aléatoirement. Par exemple, si plusieurs variables globales sont définies dans le fichier source, le compilateur peut décider de ne charger que l'adresse de la première et de référencer les autres de manière relative à celle-ci.

Ces optimisations doivent donc être désactivées si on veut pouvoir changer l'ordre des variables. Le compilateur utilisé doit permettre de désactiver sélectivement chaque optimisation, une désactivation globale de l'optimisation du code n'étant pas acceptable.

Le compilateur C *GNU* supporte l'élimination de symboles non utilisés. Cette fonctionnalité permet à l'éditeur de liens de supprimer toutes les fonctions et variables présentes dans les fichiers objets qui ne sont pas utilisées dans l'exécutable final. Pour ce mécanisme, les mêmes problèmes liés à l'optimisation peuvent s'appliquer et le compilateur ne doit pas faire d'hypothèses sur la présence d'autres symboles lorsque le code sera chargé en mémoire.

Les options de compilation adaptées à la génération de binaires hétérogènes sont donc fournies par *gcc*.

5.1.4 Implémentation de la bibliothèque

La bibliothèque *elfpp* réalisée est capable d'accomplir les tâches suivantes de façon indépendante :

– Lecture des fichiers ELF objets
– Lecture des fichiers ELF exécutables
– Désérialisation des symboles et des relocations
– Sérialisation des symboles et des relocations
– Ecriture des fichiers ELF objets

La désérialisation est faite par étapes réalisées à la demande. Lors de la lecture d'un fichier ELF, celui-ci est tout d'abord chargé de manière classique, c'est-à-dire chaque section est allouée et chargée en mémoire mais son contenu n'est pas interprété. Il est alors possible de travailler directement sur ces sections et d'écrire un nouveau fichier sans requérir plus d'assistance de la part de la bibliothèque.

Plusieurs niveaux de désérialisation peuvent être déclenchés successivement par des appels de fonctions (fig. 5.3) :

– Création des objets C++ représentant les symboles et relocations à partir des tables contenues dans les sections du fichier *ELF*.
– Création de symboles pour toutes les zones orphelines des sections.
– Allocation et copie du contenu de chaque symbole depuis la section contenante.
– Rattachement des relocations relativement aux symboles.

Dès la première étape de désérialisation, il est possible de parcourir la structure du fichier *ELF* en accédant aux objets C++ et aux structures de données en mémoire (fig. 5.4).

Ainsi, la modification du fichier *ELF* est simple et flexible, il est possible de localiser et de modifier le contenu d'un symbole mais également de le déplacer avant d'écrire un nouveau fichier en mémoire (fig. 5.4). La sérialisation a lieu avant l'écriture puisque le fichier a été désérialisé et modifié.

```
int main(int argc, char **argv[])
{
  elfpp::object obj(argv[1]);

  obj.parse_symbol_table();
  obj.load_symbol_data();
  obj.set_relative_relocs();

  ...
```

FIGURE 5.3 – Exemple de désérialisation complète d'un fichier *ELF* avec *elfpp*

5.2 L'outil de réorganisation

Pour mettre en œuvre la solution proposée dans la section 3.5.3, il convient de fusionner l'ensemble des fichiers objets construits pour un type de processeur en un seul fichier objet. Les fichiers objets obtenus pour chaque processeur sont uniformisés par l'outil de réorganisation, présenté ici, avant

```
  FOREACH(s, obj.get_section_table()) {
      std::cout << "section:" << *s << std::endl;

      FOREACH(j, s->get_symbol_table()) {
          std::cout << "symbol:" << *j->second << std::endl;

          FOREACH(k, j->second->get_reloc_table())
              std::cout << "relocation:" << *k << std::endl;
      }
  }
```

FIGURE 5.4 – Boucle d'affichage de la structure d'un fichier *ELF* avec *elfpp*

```
  /* set first byte of "var" variable to 1 */
  obj.get_section(".data").get_symbol("var").get_content()[0] = 1;

  /* move "main" function to address 32 in the section */
  obj.get_section(".text").get_symbol("main").set_value(32);

  /* write to a new ELF object file */
  obj.write(argv[2]);

  return 0;
}
```

FIGURE 5.5 – Modification et écriture d'un nouveau fichier *ELF* avec *elfpp*

d'être utilisés pour générer les binaires exécutables finaux. Le processus de compilation modulaire est détaillé section 4.1.3.

Grâce à la bibliothèque de manipulation des fichiers objets, l'outil de réorganisation reste implémenté de manière simple et son code source exprime directement les transformations à réaliser ; lesquelles sont présentées ici. Son code source est d'environ 400 lignes seulement.

5.2.1 Passe de reconnaissance

Une fois que chaque fichier objet se trouve chargé en mémoire et désérialisé, il faut effectuer un rapprochement entre les symboles équivalents provenant des différents fichiers. Le nom des symboles est utilisé comme critère de rapprochement. Ainsi, un symbole possédant le même nom dans les fichiers objets destinés à des processeurs différents, devra être relogé à la même adresse mémoire.

Seules les sections chargées en mémoire doivent être traitées. Une liste de noms de sections à traiter est chargée depuis un fichier de configuration. Des objets C++ appelés méta-sections regroupent les sections des différents fichiers objets ayant le même nom.

Dans un premier temps, une liste commune de symboles est dressée en parcourant la structure de chaque fichier objet. Un méta-symbole est créé pour regrouper les symboles aux noms identiques provenant des différents fichiers, il contient les informations suivantes :

– Le nom commun des symboles
– La liste des symboles regroupés sous ce nom

– La taille du plus grand symbole avec ce nom
– Un lien vers la meta-section contenante

Chaque méta-symbole va être relogé selon des règles différentes, dépendantes du type de la section contenante. Notons qu'un symbole spécifique à un type de processeur est également décrit par un méta-symbole ne contenant qu'une entrée.

Cas des symboles locaux

Il subsiste un problème dans le choix du nom comme identifiant unique d'un symbole pour l'ensemble du noyau et de l'application. En effet le langage C autorise l'utilisation des symboles à portée locale avec le mot clé `static`. Ainsi, rien n'empêche l'utilisation du même nom pour deux fonctions ou deux variables dans des fichiers sources différents. Il en résulte l'apparition de plusieurs symboles avec le même nom dans le fichier binaire fusionné pour réorganisation. Ceci est un comportement habituellement valide, mais l'ambiguïté introduite empêche l'utilisation du nom comme critère pour garantir un comportement équivalent de deux fonctions.

Pourtant l'uniformisation doit également concerner ces symboles puisque rien n'empêche par exemple d'affecter un pointeur sur fonction avec une fonction à portée locale. C'est même chose courante et le support de l'hétérogénéité natif doit en tenir compte.

Pour résoudre ce problème d'implémentation, il faut lever l'ambiguïté sur le nom de symboles locaux avant de fusionner chaque fichier objet provenant d'un ficher source. Une opération supplémentaire est introduite juste après la compilation et l'assemblage d'un fichier source : tous les symboles locaux sont renommés dans le fichier objet. Ils sont en fait préfixés par une chaîne identifiant le code source du fichier de manière unique.

5.2.2 Réorganisation des symboles

La réorganisation des symboles consiste à réaffecter des adresses à tous les symboles de manière identique dans tous les fichiers objets. La réaffectation des adresses est effectuée en parcourant la liste des méta-symboles, en ajoutant la taille de chaque symbole à un compteur d'adresse.

Cette réorganisation diffère selon le type de section et un comportement approprié est dicté dans le fichier de configuration pour chaque section.

5.2.3 Cas des sections de code

Les différentes fonctions contenues dans les sections de code ont des tailles variables et imprévisibles. De plus, le code d'une même fonction pour plusieurs types de processeurs différents n'est souvent pas identique.

Dans la solution mise en place, l'affectation des nouvelles adresses aux fonctions se base sur la taille de la plus grande fonction, le code des fonctions plus petites possédant le même nom étant complété avec des octets nuls.

Une optimisation envisageable de cette approche serait de ne considérer que les fonctions utilisées par les relocations d'un certain type. Selon le processeur utilisé ceci pourrait permettre de déterminer si une fonction est utilisée par un pointeur sur fonction ou non, et ainsi de n'aligner que les fonctions concernées.

Une autre optimisation consisterait à mettre en place une série de points d'entrée alternatifs à la fonction, constituée d'instructions de saut vers le point d'entrée original. Ainsi une série d'instructions de saut pourrait posséder des adresses uniformisées tandis que le corps des fonctions resterait contigu.

5.2.4 Cas des sections de données modifiables

Contrairement aux sections de code, les données doivent posséder un format rigoureusement identique pour tous les processeurs. L'outil de réorganisation vérifie que les variables possèdent la même taille et que leur contenu d'initialisation est identique, le cas échéant.

Certaines variables peuvent être spécifiques à un type de processeur et n'apparaître que dans un des fichiers objets. C'est le cas de certaines variables des couches basses et dépendantes du matériel dans l'exo-noyau. Dans ce cas la place est laissée vacante dans la section du point de vue des autres processeurs. Cependant, le chargement de données dans la mémoire partagée ne surviendra qu'une fois et ne concernera qu'un seul des fichiers binaires, choisi arbitrairement. Il faut donc s'assurer que les valeurs d'initialisation des variables spécifiques sont recopiées dans tous les fichiers afin qu'ils puissent servir indifféremment au chargement.

De ce point de vue les variables spécifiques à un processeur possédant une relocation apporte une difficulté supplémentaire. Il s'agit par exemple d'une variable qui n'existerait que pour un des types de processeurs et qui serait initialisée avec un pointeur sur un autre symbole, générant ainsi une relocation qui ajuste la valeur d'initialisation lors de l'édition des liens finale. Dans ce cas la relocation doit être recopiée plutôt que la valeur.

5.2.5 Cas des sections de données en lecture seule

Les sections de données posent un problème supplémentaire : le compilateur y stocke des données précalculées qu'il génère, par exemple des éventuelles tables de sauts pour les constructions `switch/case` du langage C. Ces données sont naturellement complètement différentes d'un processeur à l'autre. De plus, certaines données ne sont pas clairement délimitées par des symboles, de sorte qu'il n'est pas possible d'identifier leur plage d'adresse exacte.

Ces données restant spécifiques à un type de processeur, aucune réorganisation n'est cependant nécessaire. D'une manière plus générale, toutes les données en lecture seule peuvent rester dissociées puisqu'elles ne servent pas à la communication entre les processeurs.

Dans la solution mise en place, les sections de données en lecture seule ne sont pas réorganisées mais simplement chargées à des adresses différentes en mémoire partagée. Ainsi n'importe quel processeur peut y accéder et échanger des pointeurs vers ces données avec tous les autres processeurs.

Une optimisation évidente consisterait à extraire toutes les données en lecture seule ayant un contenu identique dans les différents fichiers binaires pour les factoriser en un seul endroit de la mémoire partagée.

5.3 Conclusion sur l'édition des liens pour architecture hétérogène

La maîtrise et la compréhension de la chaîne de compilation et du format de fichier binaire employé ont permis de développer un outil adapté à l'uniformisation des différents fichiers exécutables. Cette compréhension en profondeur, déterminante pour le succès du projet, est possible grâce à l'accès au code source des outils libres *GNU*, bien qu'ils restent utilisés sans aucune modification. La disponibilité d'un unique compilateur supportant les différents processeurs cibles est également déterminante.

L'outil développé ici permet de contourner les différents obstacles qui subsistent après avoir pris toutes les précautions d'utilisation des options du compilateur. Ainsi chaque symbole est identifié de manière unique par son nom et nous pouvons garantir que les symboles désignés par le même nom dans le code source, référencent les mêmes données ou le même comportement du code, quel que soit le type de processeur.

Une solution a pu être apportée à chaque problème rencontré, sans faire de concession sur les fonctionnalités du langage ou les pratiques de développement que l'on connaît dans les noyaux et les applications écrites en langage C.

Notons que le développement de la bibliothèque *elfpp* en tant que logiciel libre dissocié permet sa réutilisation dans d'autres projets. C'est le cas notamment du projet *SoCLib* où elle a remplacé la bibliothèque *GNU bfd* écrite en C.

Chapitre 6

Résultats expérimentaux

L'intérêt du support natif de l'hétérogénéité par le système d'exploitation réside dans sa flexibilité quant au déploiement d'applications parallèles :

– Aucun langage, aucune bibliothèque spécifique ou primitive de communication inter-tâches n'est imposée. De plus le code source des applications existantes, déjà parallélisées pour les systèmes multiprocesseurs classiques, peut être exploité directement.
– Le placement des tâches logicielles sur les différents types de processeurs peut être effectué, au plus tard, durant la conception et même réalisé dynamiquement, lors de l'exécution.

Toutefois, le support natif de l'hétérogénéité par le système d'exploitation n'a d'intérêt que si son impact sur les performances reste limité.

Ce chapitre présente d'abord différents tests permettant de comparer les performances de *MutekH* à d'autres systèmes d'exploitation sur des plateformes classiques. Dans un deuxième temps, on cherche à évaluer le coût en performances du support de l'hétérogénéité.

Dans un système sur puce, les facteurs de surface et de consommation entrent en jeu. Nous présentons un cas de plateforme où l'hétérogénéité permet l'optimisation de ces paramètres par l'utilisation de processeurs aux complexités différentes dans une même architecture.

Enfin, l'empreinte mémoire du noyau est étudiée pour diverses architectures de processeurs et différentes plateformes matérielles.

6.1 Evaluation du coût en perfomances de l'hétérogénéité

Toutes les précautions ont été prises lors du développement du noyau pour que le support de l'hétérogénéité soit transparent du point de vue des plateformes : les solutions apportées reposent sur la définition judicieuse de l'*API* d'abstraction du matériel *Hexo*. Aucune mise en œuvre d'algorithmes spécifiques ne supporte l'hétérogénéité dans le noyau. On peut espérer que cette transparence, nécessaire à l'exécution de bibliothèques logicielles standards, préserve également les performances et que *MutekH* reste comparable aux autres systèmes d'exploitation.

De même, l'édition des liens hétérogène, aussi complexe soit elle dans sa mise en œuvre, n'effectue qu'une permutation de symboles autrement laissés dans un ordre aléatoire par le compilateur et ne doit pas dégrader les performances.

6.1.1 Démarche

L'évaluation du coût en performances du support natif du multiprocesseur hétérogène par *MutekH* est effectuée en trois étapes :

– Comparaison des performances en monoprocesseur avec d'autres noyaux connus.
– Evaluation du *speedup* obtenu avec *MutekH* en passant en mode multiprocesseur.
– Comparaison des performances de *MutekH* avec et sans hétérogénéité.

Cette démarche permet de tirer des conclusions quant aux performances du support hétérogène dans *MutekH* en les comparant à celles des solutions non-hétérogènes connues.

6.1.2 Choix des applications

Deux applications parallélisées ont été choisies pour évaluer les performances logicielles de MutekH : Une application de décodage de flux vidéo *Motion JPEG* (*mjpeg*) et une application de tri radix (*radix*). La différence majeure entre ces deux applications réside dans le type de parallélisme employé.

Le code source des applications choisies est utilisable sur des systèmes d'exploitation standards fournissant l'*API* des *thread*s *POSIX* ; ces applications s'exécutent notamment au-dessus de *GNU/Linux* et sont préexistantes à *MutekH*.

L'application *mjpeg*

L'application *mjpeg* met en œuvre un parallélisme de type *pipeline* : il s'agit d'un graphe de tâches communicantes. Les tâches réalisent les différentes étapes du décodage vidéo. La communication entre les tâches exploite des tampons de données en mémoire partagée ainsi que les primitives de synchronisation de la bibliothèque des *thread*s *POSIX*. Les variables de condition de type `pthread_cond_*` sont très sollicitées dans cette application.

Les tâches suivantes entrent en jeu pour le coeur de l'algorithme :

– *demux* : Demultiplexage du flux MJPEG entrant pour alimenter les tâches *vld* et *iqzz*.
– *vld* : Decompression de huffman du flux mjpeg. La table de Huffman et les données compressées sont présentes dans le flux.
– *iqzz* : Application d'une table de quantification aux coefficients du domaine fréquentiel. La table et les données sont transmises dans le flux, seules les données sont compressées.
– *idct* : Transformation en cosinus inverse, procédant au calcul final du flux de pixels dans les blocs de l'image.

Les tâches suivantes sont adjointes pour le fonctionnement de l'application :

– *tg* : Génération du flux d'images *Jpeg*.

– *libu* : Transposition des macros-blocs en lignes de pixels.
– *ramdac* : Sortie vers le périphérique d'affichage.

Le graphe des tâches décrit ici est présenté figure 6.1.

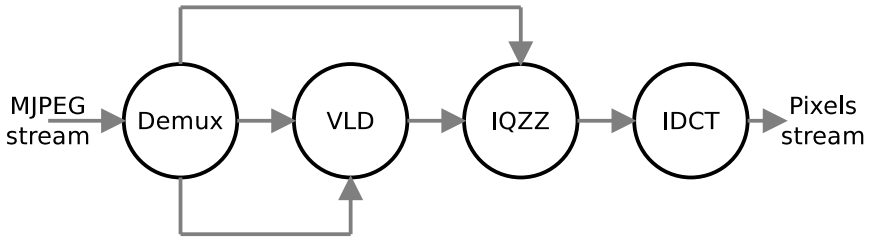

FIGURE 6.1 – Graphe de tâches de l'application de décodage *mjpeg*

Les résultats du décodage de chaque image sont vérifiés grâce à une somme de contrôle sur 32 bits.

L'application *radix*

L'application *radix* est issue de la suite de benchmarks SPLASH-2 (*Stanford Parallel Applications for Shared Memory*) [31]. Cette suite de benchmarks, proposée par l'université de Stanford pour la mesure de performances de systèmes multiprocesseurs à mémoire partagée, est largement utilisée par ailleurs.

Cette application opère un tri radix sur un tableau de valeurs numériques et met en œuvre un algorithme de type *task-farm*, où la charge de calcul est répartie ici sur plusieurs processeurs. Il est généralement créé autant de tâches que de processeurs.

Les tâches sont fortement synchronisées à l'aide de verrous et de barrières de synchronisation. Dans notre cas il s'agit des primitives `pthread_mutex_*` et `pthread_barrier_*` fournies par la bibliothèque *PThread*.

Contrairement à l'application *mjpeg*, toutes les tâches exécutent ici le même algorithme. L'application *radix* présente donc un intérêt limité en situation réelle pour un système hétérogène. Elle permet toutefois de mesurer les performances des systèmes d'exploitation pour un type de parallélisme différent de celui de *mjpeg*.

Le résultat du tri du tableau de valeurs aléatoires est vérifié en fin de calcul.

6.1.3 Evaluation sur une plateforme Mips monoprocesseur

Le premier test effectué consiste à évaluer les performances de *MutekH* dans le cadre d'une utilisation en tant que noyau embarqué sur une plateforme monoprocesseur classique.

Pour ce test, les performances de MutekH sont comparées à celles de deux autres systèmes d'exploitation embarqués bien connus et utilisés largement : *RTEMS 4.9.2* et *eCos 3.0*.

Nous cherchons ici à évaluer l'impact sur les performances de l'architecture logicielle du noyau et des techniques de l'abstraction du matériel mises en œuvre dans *MutekH*.

RTEMS est un système d'exploitation embarqué capable de temps réel dur. Tout d'abord baptisé *Real-Time Executive for Missile Systems* et destiné à des applications militaires, il est aujourd'hui disponible en tant que logiciel libre sous la dénomination *Real-Time Executive for Multiprocessor Systems*. Ce système supporte les architectures multiprocesseurs uniquement via des mécanismes de communication spécifiques de type passage de messages, mais supporte les *thread*s *POSIX* en mode monoprocesseur.

eCos est un système d'exploitation embarqué libéré par la société *Red Hat*. Le noyau est configurable et porté sur de nombreuses architectures de processeurs, mais son support des plateformes multiprocesseurs reste limité.

Ces deux systèmes permettent de lier directement l'application au noyau pour une exécution en mode noyau, c'est-à-dire sans appels système. Bien que l'architecture en exo-noyau de MutekH permette de proposer des services employant les appels système, nous avons choisi d'effectuer les mesures sans cette couche. Tous les noyaux testés proposent une implémentation de la bibliothèque *PThread*, ce qui garantit une uniformité de l'*API* pour les mesures.

Conditions du test

Les applications *mjpeg* et *radix* sont compilées au-dessus des trois noyaux configurés pour supporter une plateforme monoprocesseur *Mips*. Chaque système utilise sa configuration par défaut avec les optimisations du compilateur *GNU* activées.

Le nombre de cycles nécessaires à l'exécution de chaque application est considéré pour la mesure. Pour l'application *mjpeg*, nous mesurons le temps de décodage moyen d'une image sur un ensemble de 100 images.

Pour l'application *radix*, c'est le temps de tri d'un tableau de 262144 valeurs qui est pris en compte. L'application est configurée pour s'exécuter sur plusieurs *threads*, bien qu'un seul processeur soit utilisé ; ceci afin d'évaluer les performances du code du noyau, responsable de la synchronisation.

L'exécution des deux applications est effectuée sur une architecture matérielle *SoCLib* précise au cycle.

Résultats

	mjpeg	*radix*
MutekH	4,504 Kcycles	72 Mcycles
eCos	4,583 Kcycles	133 Mcycles
RTEMS	5,664 Kcycles	151 Mcycles

FIGURE 6.2 – Nombre de cycles d'exécution sur une plateforme monoprocesseur MIPS

Les résultats présentés figure 6.2 montrent que les performances de *MutekH* sont comparables à celles des deux autres systèmes.

Il est possible d'affirmer en première conclusion que les choix d'architecture logicielle en exo-noyau ne pénalisent pas les performances. Concernant l'application *mjpeg*, il apparaît un avantage en performances de 25% par rapport au système *RTEMS*. La différence entre *MutekH* et *eCos* n'est pas

significative. Quant à l'application *radix* où les primitives de synchronisation sont très sollicitées, l'avantage en faveur de *MutekH* est beaucoup plus important.

Il est nécessaire d'apporter quelques précisions avant d'analyser finement ces résultats :

– Ces tests mesurent globalement le nombre de cycles d'exécution. Ceci comprend : le temps de calcul par l'application, qui reste invariant entre les systèmes, représentant la majeure partie des cycles, ainsi que le temps d'exécution des appels au noyau du système d'exploitation. L'écart en performances entre les différents systèmes d'exploitation est donc beaucoup plus important que les écarts mesurés sur l'ensemble. L'impact de l'efficacité du code du noyau sur les performances globales est donc lié au degré de synchronisation requis par l'application en présence.
– Le degré de synchronisation de l'application *radix* a été artificiellement augmenté en répartissant celle-ci sur 4 *threads*, sans quoi les calculs seraient effectués séquentiellement et aucune charge ne reposerait sur le noyau.
– Le noyau *eCos* ne propose pas nativement les primitives *pthread_barrier_** qui sont très utilisées dans l'application *radix*. Celles-ci sont réimplémentées à base de verrous et de variables de condition, ce qui impacte les performances de manière significative.

Analyse fine des appels au noyau

L'instrumentation du simulateur *SoCLib* permet de surveiller le registre contenant l'adresse d'exécution du processeur et de détecter ainsi chaque saut entre les fonctions compilées dans le code binaire du noyau. Il est donc possible d'extraire de manière non intrusive une trace complète des appels et retours de fonctions pendant l'exécution.

Dans une application parallélisée, l'ordonnanceur et les primitives de synchronisation des tâches sont très sollicités. L'étude des traces d'appels de fonctions montre en effet que ces services représentent une large portion du temps d'exécution dans le noyau. Les performances de l'ordonnanceur, responsable de multiplexer le temps processeur, nous intéressent particulièrement.

Pour mesurer précisément l'efficacité de l'ordonnanceur et du changement de contexte implémentés par les différents noyaux, il est possible d'isoler et d'étudier des sections de traces passées à exécuter le code des différents noyaux. Il convient d'étudier précisément l'ensemble des sauts inter-fonctions depuis l'appel à une primitive du noyau jusqu'au retour vers une tâche de l'application.

Pour comparer les trois noyaux nous choisissons d'évaluer les deux services les plus souvent sollicités par l'application *mjpeg* : il s'agit des appels à `pthread_cond_wait` et `pthread_cond_signal`, implémentant la primitive de variables de condition. Cette dernière implique également l'utilisation des primitives de verrous `pthread_mutex_lock` et `pthread_mutex_unlock`. Ces deux cas sont plus particulièrement représentatifs des mécanismes mis en œuvre dans la synchronisation de tâches dans les systèmes logiciels d'une manière générale.

Nous considérons le même scénario pour les trois systèmes : le cas où la tâche se trouve mise en attente sur la variable de condition et réussit à prendre le verrou associé. Le scénario d'exécution de la primitive `pthread_cond_wait` se déroule à l'identique sur les trois systèmes :

– Déverrouillage du verrou de type mutex associé à la variable de condition, ce qui implique de replacer une tâche en attente sur le verrou dans sa file d'exécution (étape A).

– Mise en attente de la tâche en cours, ce qui implique l'inscription de la tâche sur une file d'attente par l'ordonnanceur (étape B) et la réalisation d'un changement de contexte d'exécution du processeur par la *HAL* (étape C). La prochaine tâche éligible à l'exécution est restaurée.
– Prise du verrou par la tâche restaurée ; on considère ici le cas où le verrou est libre (étape D).

Les figures 6.3, 6.4 et 6.5 montrent respectivement les traces des appels de fonctions concernant la primitive `pthread_cond_wait` sur *MutekH*, *eCos* et *RTEMS*. Le nombre de sauts inter-fonctions ainsi que le nombre de cycles sont indiqués ; les étapes citées sont repérées par les lettres A à D.

```
    #  cycles  fonction jump
    0  0       (pthread_cond_wait + 0)
(A) 1  223       (__pthread_mutex_normal_unlock + 0)
(A) 2  310         (sched_wake + 0)
(A) 3  441       (__pthread_mutex_normal_unlock + 0x34)
    4  560     (pthread_cond_wait + 0x70)
(B) 5  609       (sched_preempt_wait_unlock + 0)
    6  1061    (pthread_cond_wait + 0xa4)
(C) 7  1279      (cpu_context_switch + 0)
(C) 8  1529      (cpu_context_jumpto + 0)
    9  1881    (pthread_cond_wait + 0x110)
(D) 10 1946      (__pthread_mutex_normal_lock + 0)
    11 2125    (pthread_cond_wait + 0x12c)
```

FIGURE 6.3 – Liste des appels et retours de fonctions pour un appel à `pthread_cond_wait` exécuté sur *MutekH*

```
    #  cycles  fonction jump
    0  0       (pthread_cond_wait + 0x1c)
    1  93        (Cyg_Condition_Variable::wait_inner + 0)
(A) 2  360         (Cyg_Mutex::unlock + 0)
    3  849       (Cyg_Condition_Variable::wait_inner + 0x44)
    4  927         (Cyg_Thread::sleep + 0)
    5  1182          (Cyg_Scheduler_Implementation::rem_thread + 0)
    7  1527        (Cyg_Thread::sleep + 0x44)
    8  1695      (Cyg_Condition_Variable::wait_inner + 0x5c)
(B) 9  1731        (Cyg_ThreadQueue_Implementation::enqueue + 0)
    10 1914      (Cyg_Condition_Variable::wait_inner + 0x68)
    11 2019        (Cyg_Scheduler::unlock_inner + 0)
(C) 12 2379          (Cyg_Scheduler_Implementation::schedule + 0)
(C) 13 2547        (Cyg_Scheduler::unlock_inner + 0x64)
(C) 14 2724        (hal_thread_switch_context, 0 bytes above)
    15 3402        (Cyg_Scheduler::unlock_inner + 0x94)
    16 3831      (Cyg_Condition_Variable::wait_inner + 0x80)
(D) 17 4047        (Cyg_Mutex::lock() + 0)
(D) 18 4635        (Cyg_Scheduler::unlock_inner + 0)
(D) 19 5364        (Cyg_Mutex::lock() + 0x124)
    20 5571      (Cyg_Condition_Variable::wait_inner + 0xf4)
    21 5712    (pthread_cond_wait + 0x34)
    22 5736      (pthread_testcancel + 0)
    23 5985    (pthread_cond_wait + 0x3c)
```

FIGURE 6.4 – Liste des appels et retours de fonctions pour un appel à `pthread_cond_wait` exécuté sur *eCos*

```
    #  cycles  fonction jump
    0  0       (pthread_cond_wait + 0)
    1  18        (_POSIX_Condition_variables_Wait_support + 0)
(A) 2  125         (_POSIX_Mutex_Get + 0)
(A) 3  289           (_Objects_Get + 0)
(A) 4  558       (_POSIX_Condition_variables_Wait_support + 0x40)
(A) 5  625         (_POSIX_Condition_variables_Get + 0)
(A) 7  758           (_Objects_Get + 0)
(A) 8  984       (_POSIX_Condition_variables_Wait_support + 0x68)
(A) 9  1082        (pthread_mutex_unlock + 0)
(A) 10 1117          (_POSIX_Mutex_Get + 0)
(A) 11 1278            (_Objects_Get + 0)
(A) 12 1531        (pthread_mutex_unlock + 0x14)
(A) 13 1615          (_CORE_mutex_Surrender + 0)
(A) 14 1887            (_Thread_queue_Dequeue + 0)
(A) 15 1964              (_Thread_queue_Dequeue_fifo + 0)
(A) 16 2163            (_Thread_queue_Dequeue + 0x2c)
(A) 17 2326          (_CORE_mutex_Surrender + 0xa8)
(A) 18 2446        (pthread_mutex_unlock + 0x34)
(A) 19 2463          (_Thread_Enable_dispatch + 0)
(A) 20 2535        (pthread_mutex_unlock + 0x3c)
(A) 21 2567          (_POSIX_Mutex_Translate_core_mutex_return_code +
    0)
(A) 22 2603        (pthread_mutex_unlock + 0x44)
    23 2652      (_POSIX_Condition_variables_Wait_support + 0xc8)
(B) 24 2804        (_Thread_queue_Enqueue_with_handler + 0)
(B) 25 2964          (_Thread_Set_state + 0)
(B) 26 3580        (_Thread_queue_Enqueue_with_handler + 0x44)
(B) 27 3649          (_Thread_queue_Enqueue_fifo + 0)
(B) 28 3847        (_Thread_queue_Enqueue_with_handler + 0x70)
(B) 29 3960      (_POSIX_Condition_variables_Wait_support + 0x150)
    30 3977        (_Thread_Enable_dispatch + 0)
(C) 31 4050        (_Thread_Dispatch + 0)
(C) 32 4400          (_TOD_Get_uptime + 0)
(C) 33 4599            (_Timespec_Add_to + 0)
(C) 34 4744          (_TOD_Get_uptime + 0x90)
(C) 35 4794        (_Thread_Dispatch + 0x94)
(C) 36 4829          (_Timespec_Subtract + 0)
(C) 37 4917        (_Thread_Dispatch + 0xa8)
(C) 38 4935          (_Timespec_Add_to + 0)
(C) 39 5037        (_Thread_Dispatch + 0xb4)
(C) 40 5187          (_User_extensions_Thread_switch + 0)
(C) 41 5346            (_RTEMS_tasks_Switch_extension + 0)
(C) 42 5458          (_User_extensions_Thread_switch + 0x44)
(C) 43 5559        (_Thread_Dispatch + 0xfc)
(C) 44 5592          (_CPU_Context_switch + 0)
(C) 45 5971        (_Thread_Dispatch + 0x108)
    46 6276      (_POSIX_Condition_variables_Wait_support + 0x158)
(D) 47 6341      (pthread_mutex_lock + 0)
(D) 48 6359        (_POSIX_Mutex_Lock_support + 0)
(D) 49 6419        (_POSIX_Mutex_Get_interrupt_disable + 0)
(D) 50 6589          (_Objects_Get_isr_disable + 0)
(D) 51 6772        (_POSIX_Mutex_Lock_support + 0x28)
(D) 52 7080          (_POSIX_Mutex_Translate_core_mutex_return_code +
    0)
(D) 53 7128        (_POSIX_Mutex_Lock_support + 0xec)
    54 7191      (_POSIX_Condition_variables_Wait_support + 0xe0)
    55 7289    (srl_mwmr_write + 0x98)
```

FIGURE 6.5 – Liste des appels et retours de fonctions pour un appel à `pthread_cond_wait` exécuté sur *RTEMS*

La même mesure est effectuée pour la primitive `pthread_cond_signal` qui est responsable du réveil d'une tâche en attente sur une variable de condition. Cette procédure est plus simple et n'implique pas de changement de contexte d'exécution, mais simplement le transfert d'une tâche d'une liste d'attente vers une liste d'exécution.

`pthread_cond_wait`	MutekH	eCos	RTEMS
Nombre de fonctions	7	11	26
Nombre de sauts	11	23	55
Nombre de cycles	2125	5985	7289
`pthread_cond_signal`	MutekH	eCos	RTEMS
Nombre de sauts	2	11	15
Nombre de cycles	210	2574	2144

FIGURE 6.6 – Résumé des mesures de performances des ordonnanceurs des trois systèmes

Le tableau 6.6 résume les performances des différentes primitives d'après les données extraites des traces d'exécution.

Conclusion sur les performances monoprocesseur

On constate que l'architecture logicielle de *MutekH*, apporte un bénéfice important en termes de performances en plus de la flexibilité visée originellement.

L'analyse des traces d'exécution nous apprend que le compilateur *GNU* a parfaitement réalisé le travail d'intégration du code des fonctions d'*Hexo* dans les couches hautes (fonctions `inline`). La modularité logicielle apparente dans l'organisation du code source est donc complètement mise à plat lors de la compilation, ce qui permet de mettre en œuvre l'architecture en exo-noyau sans concession sur les performances.

Ceci est rendu possible grâce au placement de code dans les fichiers en-têtes, pour les fonctions les plus courtes dont le surcoût d'appel n'est pas justifié. Il convient donc de choisir soigneusement quelle fonction du noyau est placée dans un fichier source compilé de manière classique et quel code doit apparaître dans les en-têtes pour optimiser les performances sans alourdir la taille du code dans le cache du processeur. Les avancées relativement récentes du compilateur *GNU* dans ce domaine contribuent certainement à la réussite de cette approche qui a visiblement porté ses fruits dans *MutekH*. En effet, il existe beaucoup plus de fonctions source qui entrent en jeu dans les mécanismes présentés précédemment, que de fonctions apparaissant au final dans les traces d'exécution. On remarquera par exemple qu'il n'existe aucune trace des appels de fonctions aux conteneurs du module `gpct`, alors que ses listes chaînées sont en réalité massivement utilisées dans le code source de l'ordonnanceur. A l'inverse, on note que le code exécuté par *RTEMS* passe beaucoup de temps à payer le prix d'appels de fonctions afin de réaliser des opérations extrêmement simples.

A titre de comparaison, la figure 6.7 donne la trace d'exécution de `pthread_cond_wait` pour *MutekH* en désactivant l'*inlining* de code du compilateur. Le nombre de cycles augmente de 48% et le nombre de fonctions différentes, impliquées dans des appels et apparaissant dans la trace, passe de 7 à 24.

```
    #  cycles  fonction jump
    0  0       (pthread_cond_wait + 0)
    1  75        (pthread_testcancel + 0)
    2  113         (pthread_self + 0)
    3  178       (pthread_testcancel + 0x10)
    4  193         (atomic_bit_test + 0)
    5  255       (pthread_testcancel + 0x1c)
    7  286     (pthread_cond_wait + 0x20)
(A) 8  339       (pthread_mutex_unlock + 0)
(A) 9  397         (__pthread_mutex_normal_unlock + 0)
(A) 10 482           (sched_wake + 0)
(A) 11 548             (sched_queue_nolock_pop + 0)
(A) 12 613           (sched_wake + 0x2c)
(A) 13 671         (__pthread_mutex_normal_unlock + 0x34)
(A) 14 796     (pthread_cond_wait + 0x40)
    15 830       (sched_wait_unlock + 0)
(B) 16 864         (sched_preempt_wait_unlock + 0)
(B) 17 937           (__scheduler_get + 0)
(B) 18 959         (sched_preempt_wait_unlock + 0x1c)
(B) 19 1087          (sched_queue_nolock_pushback + 0)
(B) 20 1156        (sched_preempt_wait_unlock + 0x80)
(B) 21 1195          (__sched_candidate + 0)
(B) 22 1234            (__sched_candidate_noidle + 0)
(B) 23 1256              (sched_queue_nolock_pop + 0)
(B) 24 1304                (sched_queue_nolock_remove_ + 0)
(B) 25 1385              (sched_queue_nolock_pop + 0x1c)
(B) 26 1439          (__sched_candidate + 0x10)
(B) 27 1485        (sched_preempt_wait_unlock + 0x88)
    28 1548      (sched_wait_unlock + 0x10)
(C) 29 1590        (context_switch_to + 0)
(C) 30 1692          (context_leave_stats + 0)
(C) 31 1829            (cpu_cycle_diff + 0)
(C) 32 1879              (cpu_cycle_count + 0)
(C) 33 1904            (cpu_cycle_diff + 0x18)
(C) 34 2013          (context_leave_stats + 0x34)
(C) 35 2143        (context_switch_to + 0x38)
(C) 36 2160          (cpu_context_switch + 0)
(C) 37 2376          (cpu_context_jumpto + 0)
(C) 38 2731        (context_switch_to + 0x40)
(C) 39 2797          (context_enter_stats + 0)
(C) 40 2831            (cpu_cycle_count + 0)
(C) 41 2869          (context_enter_stats + 0x10)
    42 2916     (pthread_cond_wait + 0x50)
(D) 43 2936       (pthread_mutex_lock + 0)
(D) 44 2998         (__pthread_mutex_normal_lock + 0)
    45 3151     (pthread_cond_wait + 0x58)
```

FIGURE 6.7 – Liste des appels et retours de fonctions pour un appel à `pthread_cond_wait` exécuté sur *MutekH*, sans optimisation de type *inlining*

6.1.4 Evaluation du *speedup* sur une plateforme Mips

Dans *MutekH*, le support multiprocesseur hétérogène n'est pas différent du support multiprocesseur classique et le code du noyau reste configuré à l'identique dans les deux cas.

Avant d'évaluer les performances de *MutekH* sur une plateforme hétérogène, il convient donc d'éva-

luer ses performances dans un contexte multiprocesseur classique où il reste comparable à d'autres noyaux.

Il n'a pas été possible de trouver un système d'exploitation embarqué libre, largement répandu, proposant un support multiprocesseur fiable, et supportant les applications en mode noyau, afin de le comparer à *MutekH*. Les choix suivants n'ont pas été retenus pour les raisons suivantes :

– *RTEMS* ne propose pas un support du multiprocesseur en mémoire partagée.
– *eCos* propose un support multiprocesseur hautement expérimental qui n'est pas fonctionnel dans les versions testées. Cet état de fait est confirmé par la documentation et sur les listes de diffusion du projet. En outre les développeurs affirment ne pas avoir testé sur des systèmes à plus de 2 processeurs.
– *Linux* et les noyaux *BSD* ne sont pas prévus pour exécuter une application en mode noyau. Les appels système impliquent un coût en performances pouvant impacter les mesures de telle sorte que la comparaison n'est pas significative.
– D'autres systèmes, tels que *VxWorks*, dont les caractéristiques pourraient convenir, ne sont pas libres. Au-delà du problème de leur acquisition, ils possèdent souvent des licences interdisant la publication d'évaluation de leurs performances.

Nous avons donc décidé d'étudier le *speedup* (gain en performances en fonction du nombre de processeurs) obtenu avec *MutekH*, pour les applications *mjpeg* et *radix*, sur des plateformes avec 1, 2 puis 4 processeurs. Les *speedups* attendus pour le *benchmark SPLASH-2 radix* ont déjà été publiés [31], ce qui permet une comparaison.

Conditions du test

Pour ce test, nous avons choisi d'effectuer les mesures des performances de *MutekH* sur une plateforme *SoCLib* à base de processeurs *Mips*.

Les tests avec un seul processeur ont été effectués avec et sans l'activation du support du multiprocesseur à la compilation. Ceci permet d'assurer que le niveau de performances est identique à celui mesuré lors des premiers essais qui ont validé l'efficacité de *MutekH* en monoprocesseur. De cette façon il est possible de quantifier le coût de l'activation du support multiprocesseur.

Résultats

Nombre de processeurs	1 (sans smp)	1 (avec smp)	2	4
Speedup MutekH Mips, *mjpeg*	1,06	1,00	1,96	2,89
Speedup MutekH Mips, *radix*	1,00	1,00	1,99	3,94
Speedup splash-2, *radix*		1,00	1,99	3,95

FIGURE 6.8 – *Speedup* pour les applications *radix* et *mjpeg*

Le tableau 6.8 montre les *speedups* obtenus pour les deux applications et reprend les résultats publiés pour l'application *radix*.

Le *speedup* est presque idéal pour les deux applications lorsque l'on passe de 1 à 2 processeurs. Le passage à 4 processeurs pour l'application *mjpeg* montre les limites de la parallélisation de cette

application à architecture pipeline. Pour l'application *radix*, où toutes les tâches effectuent le même travail, les *speedups* obtenus sont très proches des résultats connus[31]. On note que la désactivation totale du multiprocesseur apporte un gain de 6% pour *mjpeg* tandis qu'il reste nul pour *radix*.

Les performances de *MutekH* sur une plateforme multiprocesseur classique sont donc conformes aux attentes : le support du multiprocesseur ne dégrade pas les performances.

6.1.5 Evaluation sur une plateforme multiprocesseur hétérogène

Après avoir évalué les performances de *MutekH* sur des plateformes classiques en le comparant à d'autres noyaux, il reste à évaluer le coût en performances de l'exécution sur plateforme hétérogène.

Afin de réaliser cette évaluation nous proposons de mesurer le temps d'exécution des applications sur des plateformes à 4 processeurs *SoCLib* comportant un nombre variable de processeurs *Mips* et *ARM*, numérotés de 0 à 3 :

– 4 processeurs *Mips*
– 4 processeurs *Arm*
– 2 processeurs *Mips*, 2 processeurs *ARM*
– 2 processeurs *ARM*, 2 processeurs *Mips*

Grâce à ce test il devient possible d'évaluer le coût de l'hétérogénéité : les évaluations précédentes ayant permis de montrer que l'exécution de *MutekH* sur une plateforme multiprocesseur classique possède des performances honorables, il reste à s'assurer que l'hétérogénéité de la plateforme ne fait pas chuter les performances du même code.

Lors de ce nouveau test, le nombre de cycles d'exécution des applications *mjpeg* et *radix* est évalué comme précédemment.

	4 Mips	4 ARM	2 Mips, 2 ARM	2 ARM, 2 Mips
mjpeg	25,4 Mcycles	22,6 Mcycles	25,7 Mcycles	22,4 Mcycles
radix	35,8 Mcycles	33,2 Mcycles	36,1 Mcycles	36,2 Mcycles

FIGURE 6.9 – Nombre de cycles d'exécution sur une plateforme multiprocesseur hétérogène

L'analyse des résultats présentés dans le tableau 6.9 montre que les performances des plateformes non hétérogènes basées sur les processeurs *ARM* et *Mips* sont très proches.

Pour l'application *radix* où tous les processeurs effectuent le même calcul puis se synchronisent grâce à des barrières de synchronisation, on peut s'attendre à obtenir des performances limitées par le plus lent des processeurs en présence. Les nombres obtenus indiquent que *MutekH* est plus lent de 0,8% pour cette application sur la plateforme hétérogène par rapport à la plateforme classique à base de processeurs *Mips*.

Même si les résultats montrent déjà que les différences sont minimes quant à la première application, l'analyse fine des résultats pour l'application *mjpeg* est plus délicate. En effet, tous les processeurs n'exécutent pas les mêmes calculs et certaines tâches consomment plus de temps processeur que d'autres. La migration des tâches n'est pas activée pour ce test, chaque tâche reste définitivement affectée au même processeur comme suit :

– Le processeur 0 exécute les tâches *demux* et *iqzz*.
– Le processeur 1 exécute la tâche *vld*.
– Le processeur 2 exécute la tâche *idct*.
– Le processeur 3 exécute les tâches *tg*, *libu* et *ramdac*.

Le tableau 6.10 donne le nombre de cycles passés dans chaque tâche pour le décodage de 100 images, ainsi que le nombre de cycles exécutés sur chaque processeur avec le placement énoncé. Ces valeurs sont calculées par le noyau *MutekH* et prennent en compte uniquement le temps processeur passé à exécuter du code de l'application.

	Mips	ARM
demux	73,6 Mcycles	79,8 Mcycles
iqzz	43,6 Mcycles	43,9 Mcycles
vld	120,2 Mcycles	108,4 Mcycles
idct	85,8 Mcycles	81,8 Mcycles
tg	21,6 Mcycles	19,2 Mcycles
libu	25,5 Mcycles	24,7 Mcycles
ramdac	12,2 Mcycles	11,3 Mcycles
cpu 0	117,3 Mcycles	123,8 Mcycles
cpu 1	120,2 Mcycles	108,4 Mcycles
cpu 2	85,8 Mcycles	81,8 Mcycles
cpu 3	59,4 Mcycles	55,4 Mcycles

FIGURE 6.10 – Nombre de cycles d'exécution des différentes tâches de *mjpeg*

La tâche la plus consommatrice en temps processeur sur les plateformes se révèle être la décompression de Huffman accomplie par la tâche *vld* exécutée par le processeur 1.

Le processeur ARM étant plus rapide pour l'exécution de cette tâche, il est logique que la configuration hétérogène où *vld* est exécutée sur l'*ARM* soit la plus rapide avec 22,4 millions de cycles. Même si les deux types de processeurs employés sont des processeurs généralistes, il apparaît que l'exécution de certaines tâches est légèrement plus rapide sur un des deux processeurs. Cette différence permet d'obtenir un temps d'exécution meilleur sur la plateforme hétérogène que sur la plateforme classique.

6.1.6 Conclusion sur le coût de l'hétérogénéité

Les choix réalisés pour le développement de *MutekH*, orientés vers la flexibilité du noyau et sa modularité, se sont révélés bénéfiques également pour les performances, notamment grâce à l'efficacité des compilateurs actuels.

Les performances de MutekH sur des plateformes monoprocesseurs et multiprocesseurs classiques se révèlent au moins équivalentes à celles des autres noyaux dans la classe des systèmes d'exploitation embarqués.

Les performances sur plateformes hétérogènes sont équivalentes aux performances sur plateformes multiprocesseurs classiques. Comme prévu, **le coût du support natif de l'hétérogénéité est nul**.

Cette fonctionnalité unique peut donc être exploitée afin de tirer parti du meilleur de chaque processeur dans des plateformes hétérogènes ayant un réel intérêt industriel. Elle permet la réutilisation du

code des applications existantes et l'utilisation de bibliothèques standards.

6.2 Exploitation de l'hétérogénéité d'une plateforme

Outre le support de processeurs totalement différents, le support natif de l'hétérogénéité permet avant tout l'exécution coopérative de code compilé différemment pour plusieurs processeurs. Cette fonctionnalité peut être exploitée pour cibler des processeurs de la même famille, divergeant uniquement par les extensions du jeu d'instructions, ou nécessitant des optimisations différentes de la part du compilateur.

Dans ce cadre nous proposons d'optimiser l'application *mjpeg* pour un système sur puce composé de processeurs *Mips*. Les divers processeurs *Mips* ont des caractéristiques différentes afin de s'adapter aux traitements requis par les tâches qu'ils exécutent, et d'assurer une optimisation en surface de silicium et en consommation de la puce.

Nous cherchons ici à illustrer l'intérêt du support natif de l'hétérogénéité pour optimiser les caractéristiques d'un système multiprocesseur sur puce.

Conditions du test

L'expérimentation mise en place consiste à comparer la surface de silicium et les performances atteintes pour l'application de décodage *mjpeg* lorsque tous les processeurs sont identiques mais aussi lorsque la plateforme est hétérogène.

Les tâches *idct* et *iqzz*, opérant successivement une transformation en cosinus puis l'application de coefficients, nécessitent beaucoup de multiplications. L'opérateur de multiplication à l'intérieur des processeurs peut être implémenté de différentes façons, en optimisant la surface ou la vitesse d'exécution.

Ainsi, les coeurs *Hard IP Cores*, présentés par *MIPS Technologies*, sont disponibles en deux versions proposant une optimisation en vitesse ou en surface de la *MDU* (*Multiply and Divide Unit*). Chacune de ces versions est prise en charge par le compilateur *GNU*, par une option de compilation permettant la génération de code optimal :

– Une version optimisée pour la vitesse d'exécution, opérant notamment la multiplication en 1 cycle et occupant $0.65mm^2$ de surface de silicium, prise en charge par l'option d'optimisation `-march=4km` de *Gcc*.
– Une version optimisée pour la surface, opérant la multiplication en 32 cycles et occupant $0.40mm^2$ de surface de silicium, prise en charge par l'option `-march=4kp` de *Gcc*.

Le modèle de processeur *Mips32 SoCLib* permet d'ajuster la latence des instructions pour simuler les performances de différents modèles de processeurs. Cette fonctionnalité est utilisée pour modéliser les deux coeurs présentés ici, alors que le support natif de l'hétérogénéité du noyau permet de générer un code compilé spécialement pour chaque coeur.

	3 Mips slow MDU	3 Mips fast MDU	Différence
tg	17,6 Mcycles	17,6 Mcycles	-0,06 Mcycles
demux	66,9 Mcycles	66,9 Mcycles	-0,02 Mcycles
libu	20,9 Mcycles	21,0 Mcycles	+0,08 Mcycles
vld	114,3 Mcycles	114,4 Mcycles	+0.16 Mcycles
idct	98,7 Mcycles	78,9 Mcycles	-19,70 Mcycles
iqzz	41,6 Mcycles	38,3 Mcycles	-3,36 Mcycles
ramdac	10,8 Mcycles	10,6 Mcycles	-0,14 Mcycles
Mips 0	114,3 Mcycles	114,4 Mcycles	+0.16 Mcycles
Mips 1	140,3 Mcycles	118,4 Mcycles	-21,89 Mcycles
Mips 2	116,3 Mcycles	117,3 Mcycles	+0,97 Mcycles

FIGURE 6.11 – Nombre de cycles d'exécution des tâches *mjpeg* sur différents coeurs *Mips32*

Résultats

Le tableau 6.11 permet d'étudier le coût d'exécution de chaque tâche sur les différents coeurs, et par là même de choisir la configuration matérielle et le placement logiciel le plus judicieux sur la plateforme hétérogène.

La configuration retenue est une plateforme à 3 processeurs, avec la répartition des tâches suivantes :

– Le processeur 0 exécute la tâche *vld*.
– Le processeur 1 exécute les tâches *idct* et *iqzz*.
– Le processeur 2 exécute les tâches *demux*, *tg*, *libu* et *ramdac*.

Le tableau 6.12 indique les performances du décodeur *mjpeg* et le coût en surface de silicium associé, pour trois plateformes différentes. Ces résultats sont donnés pour les deux plateformes réalisées avec des coeurs du même type, ainsi que pour la plateforme hétérogène retenue.

	3 Mips slow MDU	3 Mips fast MDU	3 Mips hétérogène
Nombre de cycles	371,0 Mcycles	350,2 Mcycles	349,1 Mcycles
Surface de silicium	1,2 mm^2	1,95 mm^2	1,45 mm^2

FIGURE 6.12 – Nombre de cycles d'exécution globale de *mjpeg* sur les différentes plateformes

Ainsi, la plateforme hétérogène, employant un seul processeur *Mips32* à multiplication rapide, obtient un niveau de performances équivalent à celui de la plateforme qui en utilise trois. Un gain en surface de silicium de 0,5mm^2, soit 25%, peut être réalisé sans dégradation des performances. Dans le cas de la plateforme non hétérogène, les opérateurs rapides restent sous-exploités pour certaines tâches, alors qu'ils sont essentiels pour la transformation en cosinus.

6.2.1 Conclusion sur l'exploitation de l'hétérogénéité

Les résultats déjà obtenus dans les sections précédentes, montrant un coût d'hétérogénéité nul, laissent entrevoir toutes les applications de la gestion native de l'hétérogénéité. En effet, les plateformes hétérogènes sont déjà répandues et leur intérêt n'est plus à démontrer dans le domaine de l'embarqué, et ce, malgré les contraintes logicielles connues jusqu'ici et résolues par *MutekH*.

L'expérimentation réalisée ici donne un exemple simple de déploiement et d'optimisation d'une application sur une telle plateforme, grâce au support de l'hétérogénéité de *MutekH*.

Une plateforme de démonstration mettant en œuvre des processeurs plus différents, tels que des processeurs disposant d'extensions du jeu d'instructions ou des processeurs aux complexités de pipeline différentes, tirerait davantage parti des différences d'optimisation du compilateur et justifierait d'autant plus la compilation séparée du code pour chaque processeur. Une telle plateforme n'était pas disponible lors des tests présentés ici.

6.3 Portabilité et empreinte du noyau

La modularité et la configurabilité du noyau permettent d'ajuster finement les fonctionnalités requises par l'application, et ainsi, de bénéficier d'une taille de code binaire réduite.

Nous présentons ici une analyse de la taille du code source des différents modules de *MutekH* ainsi que la taille du code binaire généré pour différentes configurations du noyau.

6.3.1 Taille du code source

La répartition du nombre de lignes de code source entre les différents modules présentés précédemment est donnée par le tableau 6.13. On note qu'en raison de la factorisation du code, les implémentations des bibliothèques de parallélisation (seconde partie du tableau) comportent chacune peu de lignes de code par rapport à l'exonoyau (première partie du tableau) sur lequel elles se basent

Sur l'ensemble du code développé par les contributeurs au projet *MutekH*, quelques modules sont issus de portages de bibliothèques tierces (20%, 31 797 lignes) tandis que le reste du code a été écrit sans base existante (80%, 125 426 lignes). Deux modules ont été développés et publiés en tant que logiciels libres indépendants de *MutekH* ; il s'agit de la bibliothèque de conteneurs `gpct` et de la bibliothèque de gestion de terminal `libtermui` (20 125 lignes en tout).

Le code de l'exo-noyau comprend, entre autres, l'ensemble du code spécifique à la plateforme d'une part, et au processeur d'autre part, tous deux situés dans le module `hexo` contenant la couche d'abstraction du matériel. La quantité de code source nécessaire au support d'une plateforme particulière ou d'un processeur particulier donne une bonne indication de l'effort à réaliser pour écrire un nouveau portage du noyau, tout en renseignant sur la qualité de la factorisation. Le tableau 6.14 donne le détail du nombre de lignes pour les différents processeurs et plateformes actuellement supportés. On remarque que la quantité de code source spécifique à une plateforme représente 1200 lignes en moyenne, soit 3% de l'exo-noyau seul (modules `hexo` et `mutek`), tandis que la prise en charge d'une famille de processeurs nécessite en moyenne 2100 lignes de code source, soit 7% de l'exo-noyau seul.

6.3.2 Taille des binaires générés

La taille des fichiers binaires générés est influencée par plusieurs paramètres, dictés par la configuration de l'utilisateur :

– Les fonctionnalités activées et désactivées à la compilation.

Module	Nombre de lignes
`hexo`, dépendant des plateformes	5 286
`hexo`, dépendant des processeurs	15 548
`hexo`, générique	3 178
`gpct`, conteneurs (projet indépendant)	13 527
`mutek`	5 818
`libc`	4 567
Total base	47 924
Pilotes de périphériques	22 083
`libcapsule`	1 136
`libdsrl`	1 134
`libsrl`	1 389
`libmwmr`	1 052
`libgomp` (portage)	5 875
`libpthread`	1 996
Total parallélisation	12 582
`libnetwork`	14 521
`libvfs`	7 624
`libcrypto`	758
`libelf`	4 762
`libfdt`	927
`liblua` (portage)	16 535
`libm` (portage)	9 387
`libtermui` (projet indépendant)	6 598
Total services	61 112
`examples`	18 089
Total	157 223

FIGURE 6.13 – Nombre de lignes de code source de MutekH par module

Plateforme	Nombre de lignes
`simple`	774
`emu` (user mode)	941
`ibmpc`	1 176
`soclib`	1 810
Total plateformes	5 286
Processeur	Nombre de lignes
`arm`	2 170
`mips`	2 716
`ppc`	2 144
`x86`	3 939
`x86-64-emu` (user mode on x86_64 host)	1 091
`x86-emu` (user mode on x86 host)	1 136
`avr`	1 730
Total processeurs	15 548

FIGURE 6.14 – Nombre de lignes de code spécifiques aux processeurs et aux plateformes

– Les différents algorithmes choisis, de complexité variable, parmi ceux proposés pour offrir un même service. Ceci concerne par exemple l'allocation mémoire ou la politique d'ordonnancement.
– Le type d'optimisation par le compilateur, relatif à la vitesse ou à la taille du code.

– L'architecture du processeur cible, en raison des caractéristiques de compacité propres au jeu d'instructions.
– L'utilisation du *garbage collector* de l'éditeur de liens, capable de supprimer tous les symboles qui ne sont pas utilisés lors de l'édition de liens finale.

Le nombre de configurations possibles en faisant varier les centaines de paramètres proposés par *MutekH* rend totalement impossible une évaluation exhaustive. Ainsi, seules certaines configurations sont retenues ici. Les configurations considérées pour les résultats présentés dans le tableau 6.15 sont les suivantes :

– Une configuration minimale de *MutekH*, où un maximum de services sont désactivés, avec une sélection des algorithmes les plus simples. Il est possible, dans une configuration minimale de n'activer aucune bibliothèque de parallélisation ; l'ordonnanceur de l'exo-noyau peut même être complètement désactivé. Une application simple pour microcontrôleur, sans appels système, peut par exemple utiliser directement les services de la couche d'abstraction du matériel.
– Une configuration par défaut du noyau, incluant les pilotes de périphériques courants de la plate-forme cible et employant les algorithmes par défaut du noyau, sans bibliothèque de parallélisation ou services additionnels, en dehors de la bibliothèque C standard. Cette configuration est évaluée avec et sans ordonnanceur de tâches.
– Une configuration par défaut augmentée du support de parallélisation des *thread*s *POSIX*.
– Une configuration par défaut augmentée du support de parallélisation *OpenMP*.
– Une configuration par défaut augmentée du système de fichier virtuel avec le pilote *FAT*.
– Une configuration par défaut augmentée de la pile réseau *TCP/IP*.

Les configurations de fonctionnalités précitées sont évaluées pour différentes configurations de compilation et de processeurs cibles :

– Pour une optimisation en taille (`-Os`) et en vitesse (`-O2`) par le compilateur.
– Avec et sans le support du multiprocesseur (SMP).
– Pour les cibles *SoCLib/Mips* et *Pc/x86*.

Le *garbage collector* de l'éditeur de liens est activé systématiquement. L'application, liée au noyau, comporte des appels de fonctions aux services activés pour s'assurer qu'ils ne sont pas évincés du fichier binaire.

Processeur	*Mips*			*x86*		
Configuration	`-O2`	`-Os`	SMP, `-O2`	`-O2`	`-Os`	SMP, `-O2`
minimaliste	5,3 Ko	5,3 Ko	5,3 Ko	1,9 Ko	1,7 Ko	1,9 Ko
sans ordonnanceur	30,0 Ko	30,0 Ko	34,2 Ko	25,0 Ko	19,9 Ko	27,1 Ko
avec ordonnanceur	34,2 Ko	30,1 Ko	34,3 Ko	27,7 Ko	22,0 Ko	29,8 Ko
avec *thread*s *POSIX*	38,3 Ko	34,2 Ko	42,5 Ko	30,9 Ko	24,5 Ko	33,4 Ko
avec *OpenMP*	46,7 Ko	42,6 Ko	46,8 Ko	37,2 Ko	29,8 Ko	39,6 Ko
avec système de fichiers	54,7 Ko	50,6 Ko	58,8 Ko	46,1 Ko	35,9 Ko	48,8 Ko
avec support réseau	63,4 Ko	59,3 Ko	67,6 Ko	49,9 Ko	40,4 Ko	53,0 Ko

FIGURE 6.15 – Tailles des fichiers binaires (code + données) selon plusieurs configurations

On remarque sur le tableau 6.15 que la configuration fine du noyau permet d'atteindre des tailles de binaires très faibles, au détriment des fonctionnalités. La configuration par défaut du noyau embarque l'essentiel des mécanismes système de base et l'augmentation de la taille du code reste minime

pour le support des *thread*s *POSIX*. Les autres services consomment une place proportionnelle à leur complexité, les configurations présentées ici étant celles par défaut. Chacun de ces services proposant un nombre important d'options propres, permettant de sélectionner leurs fonctionnalités et de moduler leur taille, les valeurs données dans la seconde partie du tableau sont indicatives.

6.4 Outils de mise au point

La recherche des bogues logiciels n'est pas chose facile lorsque l'on travaille sans protection mémoire, en mode noyau. Certains bugs, difficiles à trouver, peuvent rester cachés sous certaines conditions et apparaître en fonction de valeurs non définies en mémoire, par exemple. Parfois, le blocage d'une application embarquée est tout simplement dû à une taille de pile insuffisante, bien que rien ne le laisse croire à priori.

La bibliothèque *SoCLib* a permis de construire des simulateurs largement instrumentés pour la stabilisation des algorithmes et du code du noyau. Plusieurs outils de debug ont été mis au point. Ces travaux ont abouti à la conception d'une méthode générique et non intrusive d'instrumentation des *ISS* (simulateurs de jeux d'instructions), dont l'implémentation ne dépend pas de l'architecture du processeur [29]. Deux outils basés sur ces travaux sont particulièrement utiles :

– Le serveur *Gdb*, permettant l'utilisation du très répandu debugger *GNU*, outil indispensable, a été développé conjointement à *MutekH*.
– Le *MemoryChecker*, outil similaire à *valgrind* [25], largement utilisé en mode utilisateur sous *GNU/Linux*. Cet outil, dont le principe est détaillé ci-après, est en effet capable de détecter les comportements suspects du code exécuté en mode noyau, révélateurs d'erreurs de programmation. Il a été développé par Nicolas POUILLON.

Le *MemoryChecker* nécessite une collaboration entre le système d'exploitation et le simulateur. Le couple *MutekH*/*SoCLib* a permis de mener les travaux de recherche cités, tout en fournissant un puissant outil de mise au point du noyau.

Cet outil surveille les accès mémoire effectués par le processeur ainsi que certains registres pour détecter les situations improbables. Il garde une représentation de la mémoire, pour traquer l'état d'initialisation de chaque mot. Il est capable de détecter diverses opérations suspectes, telles que :

– les accès aux zones de mémoire non allouées,
– les accès en lecture aux variables non préalablement initialisées,
– les accès au-delà du pointeur de pile du contexte en cours d'exécution,
– les changements de contexte invalides,
– les allocations de mémoire non libre et la libération de mémoire non allouée,
– la non désactivation des interruptions dans les sections de code critiques,
– les prises de verrous à attente active (*spinlocks*) menant à des *deadlocks*.

6.5 Travaux annexes

MutekH est déjà utilisé en tant que système d'exploitation classique par plusieurs projets de recherche dans différents laboratoires, notamment grâce à son support de qualité pour les architectures matérielles modélisées avec la bibliothèque de simulation *SoCLib*, qui supporte toutefois plusieurs autres systèmes d'exploitation.

On peut citer quelques utilisations de *MutekH* :

- Essais et validation durant le développement de l'architecture *many-core TSAR* [1] (projet européen MEDEA). Exécution sur des plateformes à 64 processeurs en simulation *SystemC* et sur *FPGA*. Travail réalisé par Eric GUTHMULLER.
- Mise au point et validation d'une application logicielle de décodage vidéo *H264* parallélisé [8] par *Thales communication* dans le cadre du projet *SoCLib* (projet ANR). Travail réalisé par Fabien COLAS-BIGEY.
- Utilisation à l'*IRISA* dans le cadre d'un projet de recherche concernant la génération automatique de jeux d'instructions [22]. A cette occasion *MutekH* a été porté sur le processeur *Nios2*, coeur destiné aux *FPGA Altera*. Ce portage réalisé sans intervention des développeurs principaux est en cours d'intégration à la branche principale. Travail réalisé par François CHAROT.
- Utilisation dans le cadre du projet *SecBus* [30], mettant en oeuvre un composant matériel de chiffrement et vérification d'intégrité à la volée des accès à la mémoire externe dans un *SoC*. Projet réalisé en partenariat entre *Telecom ParisTech* et *STMicroelectronics*. Travail réalisé par Sebastien CERDAN.
- Utilisation pour la validation de l'architecture *Multi-compartment* [28] développée chez *STMicroelectronics*, qui permet de protéger la mémoire selon plusieurs domaines définis dans l'espace d'adressage physique. Travail réalisé par Joël PORQUET.
- Utilisation pour la validation de l'architecture *ADAM* (*Adaptive Dynamic Architecture for MP2SoCs*) [32]. Projet ANR visant à concevoir une puce massivement parallèle auto-reconfigurable pour résister aux altérations. Travail réalisé par Dimitri REFAUVELET.
- Portage sur le microcontrôleur *Atmel SAM7* à base de coeur *ARM7*, développement d'un système de fichier virtuel, par le *Lip6*. Travail réalisé par Nicolas POUILLON.
- Développement d'une pile réseau par le *Lip6*. Travail réalisé par Matthieu BUCCHIANERI.

En dehors de la simple utilisation, à ce jour, plusieurs de ces personnes ont largement contribué à divers modules du code source de *MutekH*.

6.6 Conclusion

L'évaluation de *MutekH* montre que le support natif de l'hétérogénéité des processeurs par le noyau du système d'exploitation n'implique aucune perte de performances par rapport à un système d'exploitation classique.

Les résultats obtenus laissent entrevoir toute la flexibilité de déploiement des applications existantes sur des systèmes multiprocesseurs hétérogènes, permettant de tirer le meilleur parti de chaque type

de processeur en présence, tout en exploitant des bibliothèques et des méthodes de développement standards. Les objectifs de la thèse ont donc été pleinement atteints.

La modularité et la factorisation du code permises par l'architecture en exo-noyau apportent la facilité de portage vers de nouvelles architectures et simplifie le développement de nouvelles bibliothèques de parallélisation. La configurabilité poussée du projet et le paramétrage judicieux du compilateur permettent de maîtriser la taille du code binaire généré. Cette flexibilité fait de *MutekH* un système de choix pour les projets de recherche.

Tout en démontrant la viabilité de la solution pour les plateformes hétérogènes, *MutekH* se compare aux autres systèmes d'exploitation de sa classe pour les systèmes monoprocesseurs, et offre en même temps un support multiprocesseur robuste, encore peu fréquent dans le domaine des systèmes d'exploitation embarqués libres.

Ces atouts font que *MutekH* est déjà utilisé dans plusieurs laboratoires dans le cadre de projets sans rapport avec l'hétérogénéité, et continue d'évoluer grâce à ses contributeurs.

Conclusion

Cette thèse a pour objectif la conception et la réalisation d'un système d'exploitation supportant nativement les plateformes multiprocesseurs hétérogènes à mémoire partagée. Le système présenté permet à l'application de s'exécuter nativement, et de façon répartie, sur des processeurs qui sont différents par leur architecture ou par leur jeu d'instructions. La gestion native de l'hétérogénéité permet effectivement l'utilisation d'applications existantes, conçues pour s'interfacer avec les bibliothèques de parallélisation standards les plus répandues, sans développement supplémentaire lors du portage.

Différents défis techniques ont dû être relevés quant à la réalisation du système *MutekH* pour les plateformes hétérogènes raisonnables auxquelles nous nous sommes intéréssés : Les différences entre processeurs ont pu être masquées élégamment, les solutions techniques logicielles apportées ont permis de contourner chaque contrainte liée à l'hétérogénéité, sans toutefois pénaliser les performances.

Certains travaux traitent l'hétérogénéité en intervenant à différents niveaux de la chaîne de développement du logiciel : A travers des bibliothèques et des *API* imposées, qui permettent de répartir l'application, mais aussi au niveau du langage de programmation, qui permet la parallélisation des applications sur la plateforme. Cette prise en charge non native de l'hétérogénéité pose néanmoins plusieurs problèmes. Tout d'abord, les approches qui proposent des mécanismes de communication spécifiques entre les tâches s'exécutant sur les processeurs de différents types, figent la structure du logiciel très tôt dans le cycle de conception. L'architecture du logiciel et le graphe des tâches de l'application dépendent alors du matériel, ce qui aboutit à l'écriture de code spécifique. Cette approche casse la flexibilité de déploiement des tâches sur les processeurs. De plus, ces solutions imposent plusieurs contraintes aux développeurs d'applications en exigeant une bibliothèque ou un langage dédié. Ceci limite très fortement la réutilisation de code initialement développé pour des plateformes multiprocesseurs classiques.

Nous avons montré que le support natif des plateformes hétérogènes est lié à la portabilité, au multiplexage des ressources et à l'abstraction du matériel dans le noyau du système d'exploitation. L'architecture en exo-noyau est apparue comme l'approche idéale pour supporter nativement l'hétérogénéité des processeurs. Cette architecture logicielle modulaire permet en effet la plus forte abstraction. Cette modularité du système d'eploitation est adaptée à la gestion et au multiplexage des ressources matérielles par les couches basses du noyau, tout en rendant l'hétérogénéité invisible pour les couches plus hautes. Cette architecture a permis de confiner le support de l'hétérogénéité dans l'exo-noyau, tandis que plusieurs types d'*API*, telles que les bibliothèques de parallélisation *OpenMP*, *thread POSIX* ou *Capsule*, ont pu être développées en profitant du support de l'hétérogé-

néité de façon transparente.

Nous avons donc réalisé un exo-noyau rendant possible l'exécution d'un système d'exploitation sur une plateforme matérielle multiprocesseur hétérogène. Le bénéfice apporté par un tel système est réel puisqu'il permet l'exploitation de processeurs aux spécificités et complexités différentes.

Le premier obstacle important à la réalisation concerne la multiplicité des codes binaires exécutables au sein d'une même plateforme, que les jeux d'instructions soient différents ou identiques. La virtualisation de la mémoire contenant le code, apporte une solution à ce problème, moyennant la réorganisation préalable des fonctions et variables dans les fichiers binaires. Nous avons démontré que cette technique n'impacte pas les performances lors de l'exécution.

Le second obstacle concerne les mécanismes matériels spécifiquement utilisés par le système d'exploitation, lesquels peuvent différer d'un processeur à l'autre. La couche d'abstraction du matériel *Hexo* apporte une réponse à ce problème. Cette *HAL* conçue avec soin spécialement pour les plateformes hétérogènes, n'implique pas de coût en performances supplémentaire, même lorsqu'elle est employée sur une plateforme classique.

La modularité extrême de la solution mise en place, imposée par l'architecture en exo-noyau, s'est révélée adaptée au support de l'hétérogénéité. Elle a permis de cloisonner le code spécifique à la gestion de l'hétérogénéité dans les couches basses et d'exploiter ces mécanismes pour l'implémentation d'algorithmes classiques pour tous les services de plus haut niveau du noyau. Cette approche a également permis d'apporter l'hétérogénéité native à plusieurs bibliothèques standards de parallélisation, pour couvrir un maximum d'applications existantes.

L'outil développé pour l'édition des liens hétérogène permet de contourner les différents obstacles qui subsistent après avoir pris toutes les précautions d'utilisation des options du compilateur. Ainsi il devient possible de garantir que les symboles désignés par les mêmes noms dans le code source référencent les mêmes fonctions et les mêmes variables, et ce quel que soit le type de processeur.

Les diverses mesures effectuées et comparaisons de *MutekH* avec d'autres systèmes montrent que le support natif de l'hétérogénéité des processeurs par le noyau du système d'exploitation n'implique aucune perte de performances par rapport à un système d'exploitation classique.

MutekH est plus rapide que les systèmes d'exploitation libres *eCos* et *RTEMS* sur des plateformes monoprocesseurs pour les deux applications testées, et les expérimentations montrent que la conception modulaire et configurable du système contribue à ce résultat. Le gain en performances (*speedup*) sur les systèmes multiprocesseurs classiques est comparable à celui du noyau *Linux*. Nous avons démontré que les performances ne chutent pas lorsque l'on exécute *MutekH* sur une plateforme multiprocesseur hétérogène.

Ces résultats obtenus laissent entrevoir toute la flexibilité de déploiement des applications existantes sur des systèmes multiprocesseurs hétérogènes, permettant de tirer le meilleur parti de chaque type de processeur en présence, tout en exploitant des bibliothèques et des méthodes de développement standards.

Tout en démontrant la viabilité de la solution pour les plateformes hétérogènes, *MutekH* se compare aux autres systèmes d'exploitation de sa classe, et offre aussi un support multiprocesseur robuste, encore peu fréquent dans le domaine des systèmes d'exploitation embarqués libres.

MutekH étant publié en tant que logiciel libre est déjà utilisé dans plusieurs laboratoires dans le cadre de projets divers et sans rapport avec l'hétérogénéité, et continue d'évoluer grâce à ses contributeurs.

Les objectifs de la thèse concernant la réalisation relative à l'hétérogénéité ont été pleinement atteints ; par ailleurs ces travaux ont donné naissance à un logiciel utile à différents domaines de recherche nécessitant un système d'exploitation embarqué libre avec un bon support multiprocesseur.

Perspectives

Intégration de l'hétérogénéité aux interfaces applicatives

Les bibliothèques de parallélisation standards mises à disposition du développeur d'applications n'ont pas conscience du caractère hétérogène de la plateforme. Le déploiement de l'application sur une telle plateforme implique un placement des tâches sur les processeurs appropriés pour tirer parti des spécificités de chacun. Si cette approche est adaptée à des bibliothèques comme les *thread*s *POSIX*, elle ne permet pas d'exploiter convenablement une application basée sur *OpenMP*. Il serait donc intéressant d'étendre les directives *OpenMP* afin de pouvoir désigner le type ou même la classe de processeurs utile à l'exécution d'une section parallèle de code donnée. Le développeur de l'application aurait ainsi un moyen de contrôle simple et efficace pour exploiter les atouts de différents processeurs pour le traitement de données.

Exécution sécurisée sur système multiprocesseur hétérogène

La thèse de Sylvain LEROY, encadrée par le laboratoire *Lip6* et *Thales*, intitulée "*Étude des mécanismes matériels et logiciels nécessaires à l'exécution sécurisée et simultanée de machines virtuelles communicantes sur des systèmes multiprocesseurs hétérogènes intégrés sur puce*" va débuter fin 2010. Cette thèse propose la mise en place d'un hyperviseur sur une plateforme à base de réseau sur puce équipé pour la protection et le confinement des tâches logicielles. *MutekH* est présenti pour ce projet en raison du caractère hétérogène de la plateforme matérielle.

Glossaire

ABI	*Application Binary Interface*, conventions respectées par un compilateur pour les structures de données de base, permettant l'exécution du programme dans un langage de haut niveau. Ceci concerne généralement le format de la pile, l'utilisation des registres et les conventions d'appel des fonctions.
API	*Application Programming Interface*, interface logicielle qui normalise les noms, types, prototypes des fonctions et variables, exposés par les implémentations des bibliothèques qui la respectent.
barrières	Primitive logicielle de synchronisation qui fixe le réveil d'un nombre préétabli de tâches sur un point de rencontre
big endian	Convention sur l'ordre de stockage en mémoire des octets d'un *mot* entier. La convention big endian veut que les octets de poids fort soient stockés en premier.
bus	Ensemble de signaux électroniques qui peuvent être pilotés et lus par plusieurs composants qui y sont connectés en parallèle.
bytecode	Code binaire composé d'*opcode*s destinés à être exécutés par une machine virtuelle plutôt que par un processeur matériel.
CISC	*Complex instruction set computer*, famille de processeurs ayant un jeu d'instructions complexe et vaste et dont la taille des *opcode*s est variable.
DSP	*Digital Signal Processor*, Processeur intégrant un jeu d'instructions spécifiques spécialisé dans le traitement des signaux.
ELF	*Executable and Linkable file Format*, Format répandu sous *UNIX*, contenant le code et les données, utilisé pour les fichiers exécutables ou objets.
GPU	*Graphics Processing Unit*, Processeur d'affichage graphique que l'on trouve par exemple sur les cartes graphiques des stations de travail.
HAL	*Hardware Abstraction Layer*, couche logicielle présentant une interface identique permettant l'accès à des composants matériels différents de manière transparente.
ISS	*Instruction Set Simulator*, Simlateur de jeu d'instructions.
little endian	Convention sur l'ordre de stockage en mémoire des octets d'un *mot* entier. La convention little endian veut que les octets de poids faible soient stockés en premier.

many-core	Plate-Forme multi-core où le nombre de processeurs devient suffisamment important pour soulever des problèmes de scalabilité non résolus, dans les plateformes multiprocesseurs classiques.
MMU	*Memory Management Unit*, Unité matérielle de gestion de la mémoire virtuelle
mot	Valeur numérique stockée sur plusieurs octets contigüs.
multi-core	Plate-Forme où plusieurs processeurs sont intégrés dans une même puce.
NUMA	*Non Uniform Memory Access*, architecture ou les accès mémoire sont dirigés selon l'adresse, vers différents composants mémoire aux coûts d'accès inégaux.
opcode	Suite d'octets ou *mot* codant une instruction du processeur.
OpenMP	Interface de programmation parallèle basée sur l'utilisation de directives de parallélisation dans le code.
POSIX	Norme IEEE1003 visant à standardiser un ensemble d'interfaces logicielles fonctionnant sur les systèmes de la famille *UNIX*
processus	Contexte d'exécution fourni par le système d'exploitation, qui possède son espace d'adressage mémoire propre.
PThread	Abréviation du nom de la bibliothèque des *thread*s *POSIX*, désigne également l'objet logiciel qu'est un *thread* de cette bibliothèque.
RISC	*Reduced instruction set computer*, famille de processeurs ayant un jeu d'instruction simple et réduit et dont la taille des *opcode*s est fixe.
SIMD	*Single Instruction Multiple Data*, Instruction processeur appliquant parallèlement un traitement identique sur plusieurs valeurs de même type, dans un registre.
SoC	*System on Chip*, Système complet comportant des processeurs, de la mémoire et des périphériques intégrés dans une unique puce, présent généralement dans les systèmes embarqués.
spinlock	*verrou* à attente active
symbole	Valeur nommée qui désigne une variable ou une fonction, contenues dans les sections d'un fichier objet ou exécutable.
sémaphore	Primitive logicielle de synchronisation qui maintient un compteur du nombre de réveils programmés que peut consommer une tâche.
thread	Contexte d'exécution qui partage son espace d'adressage mémoire avec les autres threads d'un même groupe. (voir *PThread*)
verrou	Primitive logicielle de synchronisation qui garantit l'exclusivité au contexte d'exécution qui le détient à un instant donné.

Bibliographie

[1] The TSAR project. Web site, Lip6/SoC, Bull.

[2] AltiVec. Web site, Freescale semiconductor.

[3] What is CUDA. Web site, Nvidia.

[4] The Cell Architecture. Web site, IBM.

[5] RTEMS 4.6.1 On-Line Library - 23.3 : Multiprocessor Communications Interface Layer, 2004.

[6] Digital Signal Processing on the Industry-Standard MIPS Architecture. Datasheet, MIPS Technologies, 2004.

[7] NEON Technology. Datasheet, ARM, 2009.

[8] Tutorial. virtual prototyping of noc-based mpsoc : Fundamentals and case studies. The 4th ACM/IEEE International Symposium on Networks-on-Chip, 2010.

[9] AMD Close to Metal Technology Unleashes the Power of Stream Computing. Press release, AMD, November 14, 2006.

[10] Jameela Al-Jaroodi, Nader Mohamed, Hong Jiang, and David Swanson. A comparative study of parallel and distributed java projects for heterogeneous systems. *Parallel and Distributed Processing Symposium, International*, 2 :0115, 2002.

[11] E. Betti, D. P. Bovet, M. Cesati, and R. Gioiosa. Operating system's support for multicore systems : current and future trends. Techincal repport, System Programming Research Group Dept. of Computer Science, System and Industrial Engineering University of Rome Tor Vergata, 2007.

[12] Gibson David and Herrenschmidt Benjamin. Device trees everywhere, 2006.

[13] Jeff Dike. A user-mode port of the linux kernel. In *ALS'00 : Proceedings of the 4th annual Linux Showcase & Conference*, pages 7–7, Berkeley, CA, USA, 2000. USENIX Association.

[14] Dawson R. Engler. *The exokernel operating system architecture*. PhD thesis, Massachusetts Institute of Technology, October 1998.

[15] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. The exokernel approach to extensibility (panel statement). In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI '94)*, page 198, Monterey, California, November 1994.

[16] Alain Greiner, Etienne Faure, Nicolas Pouillon, and Daniela Genius. A generic hardware / software communication middleware for streaming applications on shared memory multi processor systems-on-chip. *FDL'09*, 2009.

[17] Xavier Guerin and Frédéric Petrot. A system framework for the design of embedded software targeting heterogeneous multi-core socs. In *ASAP '09 : Proceedings of the 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 153–160, Washington, DC, USA, 2009. IEEE Computer Society.

[18] C. Han, M. Goraczko, J. Helander, J. Liu, B. Priyantha, and F. Zhao. Comos : An operating system for heterogeneous multi-processor sensor devices. *MSR-TR-2006-117*, 2006.

[19] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74 : Proceedings of the IFIP Congress 74*, pages 471–475, August 1974.

[20] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[21] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge : a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43(3) :287–296, 2008.

[22] K. Martin, Ch. Wolinski, K. Kuchcinski, A. Floch, and F. Charot. Constraint-driven instructions selection and application scheduling in the *DURASE* system. In *20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Boston, USA, July7-9, 2009.

[23] Rob McCammon. Wind river solutions for multiprocessing and multicore, 2007.

[24] Ross McIlroy and Joe Sventek. Hera-jvm : Abstracting processor heterogeneity behind a virtual machine. In *The 12th Workshop on Hot Topics in Operating Systems (HotOS 2009)*, may 2009.

[25] Nicholas Nethercote and Julian Seward. Valgrind : a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6) :89–100, 2007.

[26] Andreas Gal Albert Noll and Michael Franz. Cellvm : A homogeneous virtual machine runtime system for a heterogeneous single-chip multiprocessor. In *Workshop on Cell Systems and Applications*, Beijing, China, June 2008.

[27] Pierre Palatin, Yves Lhuillier, and Olivier Temam. Capsule : Hardware-assisted parallel execution of component-based programs. In *MICRO 39 : Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 247–258, Washington, DC, USA, 2006. IEEE Computer Society.

[28] Joël Porquet, Christian Schwarz, and Alain Greiner. Multi-compartment : a new architecture for secure co-hosting on soc. In *SOC'09 : Proceedings of the 11th international conference on System-on-chip*, pages 124–127, Piscataway, NJ, USA, 2009. IEEE Press.

[29] Nicolas Pouillon, Alexandre Becoulet, Aline Vieira de Mello, Francois Pecheux, and Alain Greiner. A generic instruction set simulator api for timed and untimed simulation and debug of mp2-socs. In *RSP '09 : Proceedings of the 2009 IEEE/IFIP International Symposium on Rapid System Prototyping*, pages 116–122, Washington, DC, USA, 2009. IEEE Computer Society.

[30] Lifeng Su, S. Courcambeck, P. Guillemin, C. Schwarz, and R. Pacalet. Secbus : Operating system controlled hierarchical page-based memory bus protection. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 570 –573, 20-24 2009.

[31] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs : characterization and methodological considerations. In *ISCA '95 : Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM.

[32] Zhen Zhang, Alain Greiner, and Sami Taktak. A reconfigurable routing algorithm for a fault-tolerant 2d-mesh network-on-chip. In *DAC '08 : Proceedings of the 45th annual Design Automation Conference*, pages 441–446, New York, NY, USA, 2008. ACM.