ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA
MAÎTRISE EN GÉNIE ÉLECTRIQUE
M.Ing.

PAR
Tennessee CARMEL-VEILLEUX

ADAPTATION MULTICOEUR D'UN NOYAU DE PARTITIONNEMENT ROBUSTE VERS L'ARCHITECTURE POWERPC

MONTRÉAL, LE 7 SEPTEMBRE 2011

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE :

M Jean-François Boland, directeur de mémoire
Département de génie électrique, ÉTS

M Guy Bois, codirecteur
Département de génie informatique et logiciel, École Polytechnique

M Claude Thibeault, président du jury
Département de génie électrique, ÉTS

M Bruno De Kelper, membre du jury
Département de génie électrique, ÉTS

M Daniel Roy, examinateur externe
CMC Électronique inc.

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 26 AOÛT 2011

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

En premier lieu, j'aimerais remercier mon directeur Jean-François Boland et mon codirecteur Guy Bois, qui m'ont donné une grande latitude dans la réalisation de mon projet de maîtrise. J'ai aussi apprécié leur support tout au long de ma maîtrise et en particulier après la naissance de mon fils Léon, arrivé au milieu de mon cheminement.

J'aimerais aussi remercier l'École de technologie supérieure (ÉTS), le Fonds québécois de recherche sur la nature et les technologies (FQRNT) et le Regroupement stratégique en microélectronique du Québec (ReSMiQ) pour leur support financier qui m'a permis de réaliser mes travaux.

Enfin, je voudrais remercier tout particulièrement ma conjointe Melissa. Ses encouragements et son support ont été instrumentaux à la réussite de mes travaux.

ADAPTATION MULTICOEUR D'UN NOYAU DE PARTITIONNEMENT ROBUSTE VERS L'ARCHITECTURE POWERPC

Tennessee CARMEL-VEILLEUX

RÉSUMÉ

L'utilisation de plus en plus commune de l'architecture d'avionique modulaire intégrée (IMA) a permis de réduire le poids, la taille et l'encombrement des systèmes avioniques en consolidant l'exécution logicielle de plusieurs fonctions sur un même processeur. En même temps que l'adoption accélérée de l'architecture IMA, les microprocesseurs multicoeurs ont gagné en popularité en raison de la stagnation des fréquences d'horloge des processeurs monocoeurs. Les processeurs multicoeurs promettent d'augmenter l'intégration de fonctions logicielles, mais ces derniers ne sont toujours pas acceptés en avionique, pour des raisons de complexité qui affectent la sûreté.

La technologie principale permettant l'utilisation de l'architecture IMA est le noyau de partitionnement robuste. Un noyau de partitionnement robuste permet d'isoler les applications indépendantes d'un système IMA pour prévenir la propagation des fautes. Cette isolation est réalisée par partitionnement spatial et temporel robustes. Dans ce domaine, il n'existe actuellement aucun système d'exploitation supportant les recherches sur l'évaluation de sûreté des processeurs multicoeurs.

Nous proposons dans ce mémoire d'adapter un noyau de partitionnement robuste existant, afin qu'il supporte le déploiement de partitions en parallèle sur plusieurs coeurs d'un processeur multicoeur. Nous avons analysé l'architecture d'un noyau existant, nommé XtratuM, puis nous l'avons adapté à un modèle de partitionnement robuste multicoeur. Nous avons ensuite réalisé une implémentation de cette adaptation sur un processeur PowerPC multicoeur de la famille MPC8641 de Freescale. Le prototype résultant est nommé XtratuM-PPC. Nous présentons enfin une étude de cas de l'utilisation de XtratuM-PPC avec un plan d'exécution multicoeur.

Lors des phases d'adaptation et de réalisation, nous avons identifié un ensemble de problèmes techniques affectant la sûreté des noyaux de partitionnement robuste sur les processeurs multicoeurs. Ces problèmes mettent en relief la complexité d'implémentation de ce type de système logiciel.

Nos travaux nous permettent de conclure que l'adaptation d'un noyau de partitionnement robuste monocoeur existant à une architecture multicoeur est possible. Cependant, les problèmes de sûreté qui apparaissent avec les processeurs multicoeurs demeurent non résolus. Notre prototype de noyau de partitionnement robuste multicoeur est un point de départ pour la résolution de ces problèmes dans un environnement réel.

Mots-clés : Partitionnement, processeurs multicoeurs, systèmes d'exploitation, systèmes embarqués, avionique, IMA

ADAPTATION MULTICOEUR D'UN NOYAU DE PARTITIONNEMENT ROBUSTE VERS L'ARCHITECTURE POWERPC

Tennessee CARMEL-VEILLEUX

ABSTRACT

Increasing use of the integrated modular avionics (IMA) architecture has helped reduce the size, weight and power consumption of avionics systems by consolidating multiple software functions on a same processor. At the same time as the accelerated adoption rate of IMA architectures, multi-core microprocessors have gained popularity due to stagnating performances of single-core offerings. Multi-core processors promise to increase software functions integration, but they are still not widely accepted in avionics, for reasons of complexity which affect safety.

The technological underpinning of the IMA architecture is the robust partitioning kernel. A robust partitioning kernel maintains an isolation between independant applications to prevent the propagation of faults within the system. This isolation is achieved by robust time and space partitioning. Unfortunately, there are no multicore-capable robust partitioning kernels currently available to support research in using multicore processors in IMA.

In this master's thesis, we propose to adaptat an existing robust partitioning kernel, so that it supports the deployment of partitions on several cores of a multi-core processor. To achieve this goal, we analyzed the existing XtratuM partitioning kernel and adapted it to support a multi-core robust partitioning model. We then implemented this adaptation on the Freescale MPC8641 multi-core PowerPC processor. The resulting prototype is named XtratuM-PPC. Finally, we present a case study using a multi-core execution plan on XtratuM-PPC.

During the adaptation and implementation phases, we identified a set of technical problems which affect the safety of robust partitioning kernels on multi-core processors. These problems highlight the implementation complexity of implementation of multi-core robust partitioning kernels.

Our work allows us to conclude that it is indeed possible to adapt a robust partitioning kernel from a single-core to a multi-core architecture. However, the safety-related problems that appeared due to the use of multi-core processors remain unresolved. Our multi-core robust partitioning kernel prototype is thus a starting point for the resolution of these problems in a real-world environment.

Keywords: Partitionning, multicore processors, operating systems, embedded systems, avionics, IMA

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES EXTRAITS DE CODE

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

AECC	Airlines Electronic Engineering Committee
AMP	Asymetrical Multiprocessing
APEX	ARINC-653 Application Executive
ARINC	Aeronautical Radio Incorporated
DMA	Direct Memory Access
ESA	European Space Agency
FAA	Federal Aviation Administration
FBW	Fly-by-wire
FIFO	First-In First-Out
FMS	Flight management system
FPU	Floating-point unit
GPL	GNU Public License
IEEE	Institute of Electrical and Electronics Engineers
IPC	Inter-partition communication
ISS	Instruction Set Simulator
IVHM	Integrated Vehicle Health Management
JTAG	IEEE Joint Test Action Group
MMU	Memory Management Unit
MPU	Memory protection unit
NASA	National Aeronautics and Space Administration

NUMA	Non-uniform Memory Access
OS	Operating System
PCI	Peripheral Component Interconnect
PCS	Plan de configuration du système
RSW	Resident Software
RTCA	Radio Technical Commission for Aeronautics
SMP	Symetrical Multiprocessing
TBC	Timebase counter
TLB	Translation Lookaside Buffer
VP	Virtual Platform
VPI	Virtual Platform Instrumentation
WCET	Worst case execution time

INTRODUCTION

Depuis le début des années 2000, un nouveau modèle d'intégration avionique a pris son essor dans les derniers modèles des grands avionneurs. Le modèle d'avionique modulaire intégrée [68] (de l'anglais « Integrated Modular Avionics » ou IMA) propose de consolider et d'intégrer les fonctions avioniques des appareils dans un nombre réduit de boîtiers en exécutant plusieurs applications différentes sur un même microprocesseur.

Cette consolidation de plusieurs applications avioniques sur un microprocesseur va à l'encontre de la pratique traditionnelle. Avec le modèle d'avionique fédérée traditionnel, un ordinateur embarqué avec ses fonctions associées (alimentations, mémoires, interfaces) est dédié à chaque application avionique. Par exemple, le système de gestion de vol est réalisé avec un ordinateur complètement indépendant de celui du contrôle de climatisation de la cabine. Le modèle fédéré limite la complexité, ce qui simplifie la certification de sûreté. Cependant, l'augmentation du poids, de l'espace et de l'énergie consommée qui découle de la duplication des ordinateurs est trop élevée dans un contexte d'augmentation du nombre de fonctions et de diminution des coûts d'exploitation des avions. Le modèle IMA est de plus en plus employé dans les appareils développés récemment, dont le Boeing 787 Dreamliner et l'Airbus A380 [55].

La séparation des fonctions inhérente au modèle fédéré a cependant l'effet bénéfique de contenir les fautes. Par exemple, le plantage du logiciel dans un boîtier ne pourrait en aucun cas se propager à un autre boîtier. En contraste, dans le modèle IMA, on doit employer une isolation logicielle entre les différentes fonctions avioniques qui s'exécutent sur un même processeur afin d'empêcher la propagation des fautes entre les applications [57]. Cette isolation est réalisée à l'aide d'un mécanisme appelé « partitionnement robuste ». La norme ARINC-653 [12] spécifie un modèle de partitionnement robuste standardisé et des interfaces logicielles qui permettent de faire communiquer les applications d'un système IMA entre elles. Cette norme est

déjà supportée par plusieurs systèmes d'exploitation du domaine commercial, dont VxWorks 653 de Wind River, Integrity-178B de Green Hills, PikeOS de SYSGO et LynxOS-178B de LynuxWorks.

Jusqu'à présent, les systèmes IMA déployés par les avionneurs emploient des ordinateurs monoprocesseurs classiques. Cela peut sembler surprenant puisque les ordinateurs à processeurs multicoeurs sont actuellement la norme. Les processeurs multicoeurs ont été largement adoptés en raison du plafonnement des fréquences d'horloge et de l'augmentation du nombre de transistors intégrables sur une même puce prévu par la loi de Moore [13]. Ils sont aussi de plus en plus employés dans les systèmes embarqués dans plusieurs domaines industriels [17, 24]. Malgré cela, l'utilisation des processeurs multicoeurs en avionique en général et en IMA en particulier relève encore du domaine de la recherche. Cela est dû au fait que la complexité additionnelle des systèmes parallèles introduit plusieurs nouveaux problèmes techniques qui affectent la certification de sûreté matérielle et logicielle [37]. D'ailleurs, la norme ARINC-653 ne spécifie que le comportement d'un système monoprocesseur (monocoeur) classique.

Certains systèmes d'exploitation commerciaux permettent déjà le partitionnement robuste sur plusieurs coeurs, dont QNX Neutrino [36] et SYSGO PikeOS [62]. Le système d'exploitation PikeOS est même certifié pour utilisation avionique. Cependant, l'adoption de ces systèmes d'exploitation est ralentie par les obstacles à la certification de sûreté rencontrés par les développeurs d'applications avioniques et les avionneurs. De plus, ces systèmes d'exploitation ne sont pas ouverts, empêchant ainsi une évaluation poussée par les chercheurs.

L'exploitation des nouveaux processeurs multicoeurs pour les applications IMA est un domaine de recherche encore très fertile. À notre connaissance, aucun système d'exploitation à partitionnement robuste supportant les processeurs multicoeurs n'est présentement disponible dans le domaine académique sans licence commerciale. Cette absence d'options dans le

domaine est un obstacle à l'avancement des recherches sur les sujets liés au partitionnement robuste avec les processeurs multicoeurs.

Dans ce contexte, nous identifions le problème de recherche suivant : « Comment serait-il possible d'adapter un noyau de partitionnement robuste monocoeur existant afin de le rendre compatible à un environnement multicoeur ? »

Le développement d'une solution à ce problème permettra de mettre en relief les obstacles qui empêchent actuellement l'exploitation en pratique des processeurs multicoeurs dans le domaine de l'IMA.

Nous visons aussi, à travers la réalisation du prototype de solution, l'évaluation d'une plateforme virtuelle pour le développement logiciel en avionique. Des projets de recherche récents, mandatés par la FAA, suggèrent l'évaluation des plateformes virtuelles afin d'améliorer la qualité de vérification des logiciels embarqués avioniques complexes [43]. Nous avons donc choisi de suivre le sillage de ces chercheurs afin de valider, dans notre contexte, l'exploitation de la même plateforme virtuelle, soit Wind River Simics, pour faciliter le développement de notre prototype.

Ce mémoire présente l'adaptation logicielle du noyau de partitionnement robuste monocoeur académique « XtratuM » [19] qui vise l'architecture SparcV8 [60] afin de le rendre compatible avec un modèle de processeur multicoeur. L'adaptation ainsi réalisée, nommée « XtratuM-PPC », vise les processeurs multicoeurs de l'architecture PowerPC [34, 53]. Nous proposons une adaptation du modèle de partitionnement robuste de XtratuM applicable dans le cas des processeurs multicoeurs. De plus, nous identifions, à travers notre implémentation, des problèmes potentiels liés à l'exploitation des processeurs multicoeurs dans le contexte du partitionnement robuste. Enfin, nous présentons les résultats d'une étude de cas expérimentale de l'utilisation de notre noyau adapté, XtratuM-PPC, pour l'intégration d'un nombre d'applications plus élevé que ce qui est possible sur un processeur monocoeur.

Le corps de ce mémoire est structuré comme suit :

- le chapitre 1 présente la problématique de recherche ;

- le chapitre 2 présente une revue de la littérature pertinente au problème ;
- le chapitre 3 présente l'architecture du noyau XtratuM original ;
- le chapitre 4 présente notre adaptation multicoeur de chacun des blocs du noyau ;
- le chapitre 5 présente les détails techniques de l'implémentation de XtratuM-PPC sur les processeurs PowerPC et une étude de cas de son utilisation.

Nous concluons finalement avec une synthèse de nos travaux et des pistes de recherches ultérieures.

CHAPITRE 1

PROBLÉMATIQUE

1.1 Problème de recherche

Nous posons le problème de recherche avec la question suivante : « Dans le contexte des systèmes IMA, comment serait-il possible d'adapter un noyau de partitionnement robuste monocoeur existant afin de le rendre compatible à un environnement multicoeur ? »

Le problème est motivé par la stagnation des performances des processeurs monocoeurs depuis le milieu des années 2000 [13], ce qui force l'industrie avionique à considérer l'éventuelle adoption des processeurs multicoeurs [37]. Dans le cadre du projet CRIAQ AVIO-509, des partenaires industriels du domaine de l'avionique sont intéressés à l'utilisation éventuelle des processeurs multicoeurs pour augmenter le niveau d'intégration des plateformes avioniques afin d'en réduire les coûts. Le problème posé ci-haut a été identifié comme une première étape dans cette direction lors de réunions des chercheurs de notre équipe de projet.

1.2 Objectifs

Dans le cheminement vers la solution du problème de recherche, trois objectifs sont visés.

À notre connaissance, il n'existe actuellement aucun prototype librement disponible permettant l'évaluation du partitionnement robuste sur processeurs multicoeurs pour les applications avioniques.

1. Le premier objectif consiste à adapter un noyau de partitionnement robuste existant à une architecture multicoeur.

Le noyau adapté devra permettre :

- la consolidation d'un plus grand nombre d'applications avioniques sur une même machine en exploitant des processeurs multicoeurs existants ;
- l'exploration des principaux modèles de partitionnement temporels parallèles (AMP, SMP et hybride) ;
- le prototypage d'applications IMA embarquées dans un environnement logiciel comparable aux solutions commerciales existantes.

L'implémentation d'un noyau de partitionnement robuste multicoeur vise à rendre possible l'intégration sécuritaire d'un nombre plus élevé de fonctions avioniques dans chaque boîtier physique. La mise en oeuvre d'une implémentation multicoeur du partitionnement robuste permettra de relever des problèmes d'intégration et de sûreté encore inconnus ou encore non validés expérimentalement.

2. Le second objectif consiste à identifier des problèmes d'intégration et de sûreté découlant du partitionnement robuste multicoeur.

Les noyaux de partitionnement robuste font partie de la classe logicielle des systèmes d'exploitation. Ils doivent prendre le contrôle total du matériel afin d'isoler adéquatement les applications. Les systèmes d'exploitation embarqués sont notoirement difficiles à développer, car les bogues surviennent le plus souvent à un niveau où l'utilisation de débogueurs est difficile, voire impossible et où les conséquences de bogues sont souvent le plantage dur de la machine. L'utilisation de plateformes virtuelles a été suggérée dans la littérature [43] pour simplifier le développement et la validation des logiciels embarqués.

3. Le troisième objectif consiste à implémenter la totalité du prototype sur une plateforme virtuelle afin d'évaluer les avantages et inconvénients de ce type d'outil pour le prototypage et le test de logiciels avioniques complexes.

1.3 Contributions

Nous identifions ici les contributions découlant de la réalisation de nos travaux de recherche et de l'atteinte des objectifs du projet.

En premier lieu, nous réalisons un prototype de noyau de partitionnement robuste multicoeur adapté à partir d'un noyau existant. Le code source, la documentation et les exemples découlant de la réalisation de cette adaptation pourront servir de base à des travaux de recherche ultérieurs qui nécessitent un noyau de partitionnement robuste multicoeur fonctionnel.

En second lieu, nous présentons les problèmes d'adaptation que nous avons relevés, ainsi que la description du processus d'adaptation. Le contenu du présent mémoire sera utile à nos successeurs qui voudront réaliser le même genre d'exercice dans un environnement industriel.

En dernier lieu, nous présentons une étude de cas de l'utilisation de notre noyau adapté. Cette étude de cas est réalisée sur la plateforme virtuelle Simics. Par ailleurs, nous avons développé une méthode d'instrumentation nouvelle vouée aux plateformes virtuelles lors du débogage de notre implémentation du noyau XtratuM-PPC. Cette méthode, nommée VPI (« Virtual Platform Instrumentation »), a fait l'objet d'un article de conférence présenté à l'« IEEE International Symposium on Rapid System Prototyping » (RSP 2011). L'article est reproduit à l'annexe II. Notre couverture de l'utilisation de plateformes virtuelles sera donc réduite dans le présent mémoire, car ces résultats ne sont pas nécessaires à la compréhension de nos contributions principales liées à l'adaptation du noyau XtratuM.

1.4 Hypothèses et suppositions

Nous identifions dans cette section les hypothèses et suppositions générales qui ont servi à guider la méthodologie de recherche.

Notre hypothèse centrale est que l'adaptation multicoeur d'un noyau de partitionnement robuste existant est possible. De plus, nous émettons l'hypothèse que cette adaptation pourra supporter des plans d'exécution parallèles symétriques (SMP), asymétriques (AMP) et hybrides (SMP et AMP).

Ces hypothèses sont basées sur l'analyse préalable des architectures des systèmes d'exploitation Linux et RTEMS. Linux et RTEMS supportent déjà des modèles de programmation parallèles. De plus, le système d'exploitation temps réel RTEMS a déjà fait l'objet d'une adaptation afin de supporter le partitionnement robuste selon la norme ARINC-653 sur des processeurs monocoeurs. Cette adaptation avait été réalisée par l'équipe du projet AIR (« ARINC-653 in RTEMS ») [56], sans support de traitement parallèle. Enfin, le noyau de partitionnement robuste de PikeOS supporte maintenant les processeurs multicoeurs [30, 62]. Ce dernier est apparu vers la fin de nos travaux. Il est donc raisonnable de croire qu'il sera possible d'adapter un noyau de partitionnement robuste monocoeur à un modèle multicoeur dans nos travaux.

Afin de circonscrire plus précisément notre problématique, nous soulignons les suppositions suivantes :

- La cible des développements expérimentaux est l'architecture PowerPC. Plusieurs processeurs multicoeurs commerciaux emploient déjà cette architecture.
- La plateforme matérielle modélisée est considérée comme fiable et déterministe au sens de ses spécifications. Nous n'évaluerons pas la fiabilité intrinsèque de la plateforme matérielle cible.
- La plateforme virtuelle employée pour les travaux de prototypage est supposée fiable et déterministe.
- Aucune contrainte de certifiabilité n'est appliquée aux développements expérimentaux, malgré la considération de ces mêmes contraintes dans la résolution du problème posé.
- Le modèle de partitionnement robuste défini par la norme ARINC-653 est prépondérant en industrie, mais nos travaux d'adaptations ne viseront que le respect de l'esprit de la norme et non pas son implémentation.

Ces suppositions générales s'appliquent en général à l'intégralité de nos travaux. Ces dernières s'ajoutent aux suppositions propres à chacune des tâches d'adaptation, qui seront présentées dans leur contexte particulier.

CHAPITRE 2

REVUE DE LA LITTÉRATURE ET FONDEMENTS

2.1 Survol

Dans ce chapitre, nous présentons les fondements nécessaires à la contextualisation du reste du mémoire. En premier lieu, nous décrivons le type de systèmes avioniques ciblés par nos travaux. En second lieu, nous jetons les bases du domaine du partitionnement robuste et présentons les méthodes employées pour implémenter les noyaux de partitionnement robuste modernes. Par la suite, nous justifions le choix du noyau de partitionnement robuste XtratuM comme base pour l'adaptation multicoeur décrite aux chapitres 4 et 5. Finalement, nous fournissons quelques références importantes à propos des processeurs multicoeurs.

2.2 Systèmes embarqués avioniques

Notre travail de recherche s'inscrit dans le domaine des systèmes embarqués avioniques.

Le domaine de l'avionique regroupe la conception, la réalisation et l'entretien des systèmes électriques, électroniques et informatiques des avions. Dans un avion de ligne moderne, l'avionique comprend entre autres une multitude de systèmes embarqués qui collabore pour assurer le bon fonctionnement de l'appareil.

Dans les sections suivantes, nous présentons un survol des caractéristiques principales des systèmes embarqués avioniques. En premier lieu, nous présentons des exemples de fonctions avioniques basées sur des systèmes embarqués. Ensuite, nous discutons de la sûreté des systèmes embarqués avioniques. Finalement, nous introduisons l'architecture des systèmes embarqués avioniques, incluant le modèle d'avionique modulaire intégrée et son rôle dans les systèmes avioniques contemporains.

2.2.1 Fonctions avioniques

Le cockpit moderne est occupé dans la majorité des cas par deux professionnels : un pilote et un copilote. Les postes de mécanicien de bord et de navigateur ont pour ainsi dire disparu depuis l'apparition de systèmes avioniques suffisamment performants pour les remplacer [61, p. 172]. Ainsi, le contrôle des paramètres mécaniques et du plan de vol est maintenant réalisé par des systèmes embarqués. Ces derniers mesurent et évaluent en temps réel une multitude de capteurs. Les conditions normales sont gérées automatiquement et l'attention du pilote n'est demandée que lorsqu'un évènement anormal survient. De cette façon, la surveillance continuelle d'un mur complet d'indicateurs n'est plus nécessaire. Les pilotes peuvent donc se concentrer uniquement à leurs tâches principales de pilotage et de gestion du vol, avec un nombre réduit de distractions.

Deux exemples de fonctions avioniques très communes sont les systèmes de commande de vol électriques (de l'anglais « fly-by-wire », abrégé FBW) et les systèmes de gestion de vol (de l'anglais « flight management system », abrégé FMS).

Les systèmes de commande électrique de vol contrôlent le déplacement de toutes les gouvernes (ailerons, déporteurs et volets) à l'aide d'actionneurs électriques en fonction des commandes du pilote et de lois de contrôle adaptées [16, pp. 24-2–24-3], au lieu d'employer des systèmes à câbles à lien direct. Les systèmes FBW améliorent la stabilité statique de l'appareil et garantissent la sûreté des paramètres de l'enveloppe de vol en prévenant par exemple une inclinaison latérale excessive [26, pp. 140–142].

Les systèmes de gestion de vol, quant à eux, remplacent la position du navigateur dans le cockpit par un ensemble d'ordinateurs et d'affichages. Le FMS fournit aux pilotes les données de trajectoire nécessaires au respect du plan de vol et leur permet de suivre en temps réel l'évolution de la position de l'appareil [67].

2.2.2 Sûreté des systèmes avioniques

Le droit de vol dans l'espace aérien public implique des appareils sûrs. La sûreté est certifiée par les autorités d'aviation civile, telles que la « Federal Aviation Administration » (FAA) aux États-Unis et Transport Canada. Ces certifications ont pour objectif d'assurer la sûreté du public, des passagers et des équipages [9, p. 145].

2.2.2.1 Fiabilité des systèmes avioniques

Selon McIntyre de la FAA [47, p. 327], la sûreté des avions passe par la fiabilité et c'est pourquoi la réglementation vise principalement cet aspect. Deux éléments sont nécessaires à l'obtention de la fiabilité : 1) la conception à sécurité intégrée (« fail-safe design ») et 2) la redondance des systèmes. Selon Allerton, ces deux concepts sont inextricablement reliés [9, p. 146]. Il est actuellement impossible de réaliser des systèmes électroniques ou mécaniques dont la probabilité de défaillance respecte le barème actuel d'une défaillance par 10^9 heures [1]. La redondance des systèmes est le seul moyen d'obtenir la fiabilité nécessaire.

Les moteurs, boîtiers avioniques, actionneurs et autres composants sont dits « certifiables » lorsqu'ils ont passé une évaluation de conception et passé leurs tests de certification. À partir de ce moment, ils peuvent être inclus dans un système intégré, qui sera certifié dans son ensemble.

2.2.2.2 Lignes directrices de certification

Des lignes directrices (« guidelines » en anglais) ont été développées par les organismes régulateurs, de concert avec l'industrie, afin d'assurer l'uniformité et la faisabilité commerciale

1. Une défaillance par 10^9 h équivaut à une défaillance par 1 000 ans pour une flotte de 100 avions ou une défaillance par 100 000 ans pour un seul avion [9, p. 145].

des critères d'évaluation lors de l'étape de certification. Ces documents couvrent tous les aspects du développement d'équipements pour les avions, incluant les systèmes mécaniques, électroniques, logiciels, ainsi que l'intégration de ces derniers.

Les lignes directrices sont des « normes » de facto dont la validité est généralement reconnue par les organismes régulateurs à travers le monde. En prouvant avoir suivi le cheminement d'une ligne directrice, on prouve que notre système atteint les objectifs de sa conception [33].

Il est important de noter que les lignes directrices sont très vagues et ne sont pas des normes en tant que telles. Chaque fournisseur doit développer ses propres processus de développement pour atteindre les objectifs de certification. Le respect de l'esprit des lignes directrices assure cependant une plus grande probabilité que les processus développés respectent les standards de certification. Ces standards de certification sont disponibles auprès des autorités aériennes [8].

Les lignes directrices les plus populaires en avionique sont développées par la « Radio Technical Commission for Aeronautics » (RTCA), un organisme sans but lucratif. Plusieurs normes avioniques sont développées par l'« Airlines Electronic Engineering Committee » (AECC) et publiées par Aeronautical Radio, inc. (ARINC). Ces normes sont respectées par l'industrie et reconnues par les organismes régulateurs.

2.2.2.3 Directive DO-178B visant le développement logiciel

La directive RTCA DO-178B [1] concerne le processus de développement logiciel en avionique. Son titre complet est « Software considerations in airborne systems and equipment certification ». Il s'agit de la plus importante directive dans le domaine du développement logiciel sur systèmes embarqués avioniques. Elle couvre tous les aspects importants pour la certification d'équipement comportant une composante logicielle :

- la planification du projet ;
- le développement logiciel ;

- la validation et vérification ;
- la gestion de la configuration (autant au niveau externe qu'au niveau des livrables).

L'esprit de la directive est le déterminisme, c'est-à-dire que chaque effet réactif du système a une cause prédéfinie, identifiable et reproductible dans les mêmes conditions [33, p. 15]. De plus, chaque fonction ou groupe d'instructions doit être traçable vers une spécification fonctionnelle. Il ne doit donc y avoir rien qui puisse être laissé au hasard ou inséré inutilement dans le logiciel.

Avant qu'un système logiciel ne puisse être certifiable, on doit s'assurer qu'il est correct. Avec DO-178B, on doit s'assurer de la préservation sémantique entre le code source et le code binaire. Les résultats des compilateurs doivent être vérifiés. Une autre contrainte de DO-178B est que le code doit être robuste aux valeurs aberrantes. En effet, si une variable contient une mesure physique impossible pour le mode d'exécution courant, l'erreur doit être détectée. Cette contrainte réduit considérablement les risques de catastrophes dans les cas de fautes logiques ou de corruption de mémoire, mais augmente grandement le nombre de cas à vérifier.

Les étapes de vérification sont les plus coûteuses dans un processus DO-178B en raison des contraintes de vérification qui sont très sévères [9]. Malgré les contraintes sévères, DO-178B est une directive pragmatique, développée conjointement par l'industrie et les organismes régulateurs. Des justifications adéquates permettent souvent de simplifier les vérifications nécessaires, ce qui permet de réduire les coûts de développement.

2.2.3 Architecture des systèmes embarqués avioniques

2.2.3.1 L'avionique fédérée

Le modèle avionique classique est le modèle dit « fédéré » : l'ensemble avionique d'un appareil est constitué d'une pluralité de systèmes conçus indépendamment par différents fabricants

et intégrés par l'avionneur. L'architecture des systèmes fédérés fournit à chaque fonction avionique les ressources suivantes [69, p. 12-2] (voir aussi figure 2.1) :

- un système embarqué dédié ;
- une infrastructure dédiée (alimentations, refroidissement, châssis) ;
- des interfaces d'entrées/sorties dédiées ;
- un bus système dédié.

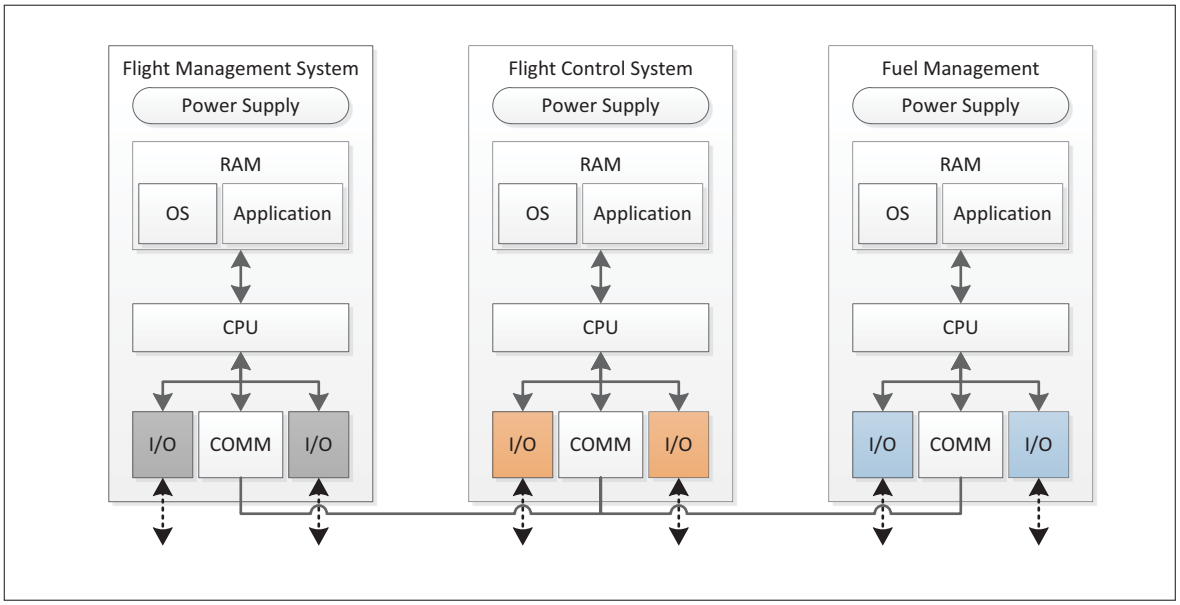


Figure 2.1 Exemple schématique d'une architecture fédérée

Le modèle fédéré facilite la conception et la certification, car les interactions avec les autres systèmes sont bien circonscrites et la complexité des systèmes est maîtrisable avec des méthodologies éprouvées depuis plusieurs décennies. L'hétérogénéité des fonctions avioniques dans un environnement fédéré augmente cependant les frais d'exploitation des appareils, en raison notamment du nombre élevé de pièces de rechange à gérer, de l'encombrement et du poids « mort » dédié à une réplication de modules à fonction unique.

2.2.3.2 L'avionique modulaire intégrée (IMA, « Integrated Modular Avionics »)

Par opposition au modèle fédéré, le modèle avionique modulaire intégré vise à faire converger sur une même plateforme matérielle plusieurs fonctions avioniques qui auraient été développées sur des systèmes indépendants dans le passé [10]. La figure 2.2 présente l'intégration des fonctions de la figure 2.1 avec le modèle IMA.

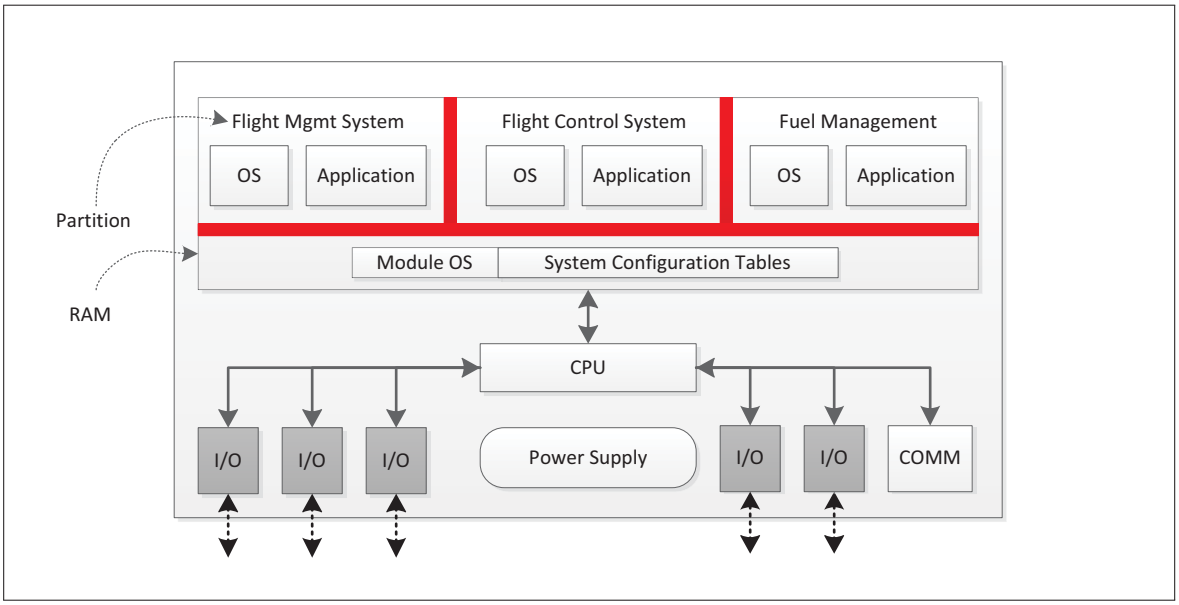


Figure 2.2 Exemple schématique d'une architecture IMA

L'objectif de cette convergence est très clair, selon ARINC651 :

> « The [IMA] system design should make maximum use of shared resources to reduce resource duplication to a minimum. Such integration lowers the cost of ownership by reducing the acquisition cost, spares requirements, weight, and volume of the avionics equipment. » [10, p. 3]

Les systèmes IMA atteignent cet objectif en fournissant :

- un système embarqué partagé où les applications sont isolées avec un partitionnement robuste (voir section 2.3 page suivante) ;
- une infrastructure commune (alimentations, refroidissement, châssis) ;

- des interfaces distantes d'entrée/sortie partagées ;
- un bus système distribué, par exemple un bus Ethernet redondant à haut débit.

L'IMA permet aussi d'améliorer la standardisation des modules matériels. Cette standardisation vise à réduire le temps de développement d'un système complet, car la phase de développement matériel n'est pas à refaire. La même équipe peut ainsi entreprendre d'autres projets sans réapprendre les spécificités d'une architecture unique au projet [10, p. 47]. De plus, la réutilisation entre différents projets de fonctions avioniques déjà programmées est un objectif à long terme de l'IMA.

La plateforme Genesis [68]développée par Smiths Aerospace est un exemple de modèle idéalisé de plateforme IMA. Le « Common Core System » (CCS), une implémentation de la plateforme Genesis, est à la base de l'ensemble avionique du Boeing 787 Dreamliner [68]. Le CCS supporterait entre 80 et 100 applications (une fonction avionique est composée d'une ou plusieurs « applications ») [6]. L'Airbus A380, quant à lui, supporterait une vingtaine de fonctions avioniques sur sa plateforme IMA développées par une dizaine de fournisseurs [35, p. 25]. Dans ces deux cas, un réseau temps-réel ARINC-664 [11] réalise l'interconnexion entre les modules IMA.

L'intégration IMA ne respecte les critères de sûreté logicielle que si les développeurs ont l'illusion d'avoir un système complètement indépendant et isolé. Sans cette indépendance entre les logiciels, une défaillance logicielle peut se propager à l'intérieur du système.

Les nouveaux systèmes d'exploitation adaptés à l'IMA sont dotés de mécanismes de partitionnement robuste qui isolent les partitions avec un niveau de sûreté très élevé. Nous présentons un survol de ces mécanismes à la section suivante.

2.3 Partitionnement robuste

La contribution principale de ce mémoire est l'adaptation d'un noyau de partitionnement robuste afin qu'il puisse supporter les processeurs multicoeurs. Dans cette section, nous défini-

rons en premier lieu ce qui est entendu par « partitionnement robuste », puisqu'il s'agit d'un concept central à nos travaux. Par la suite, nous introduirons les considérations techniques liées au partitionnement robuste, incluant un survol de la norme ARINC-653. Finalement, nous présenterons une vue d'ensemble des différentes méthodes de partitionnement robuste.

2.3.1 Définition du partitionnement robuste

Le partitionnement robuste est défini comme suit dans la norme ARINC-653 :

> « *[Robust Partitioning :]* A mechanism for assuring the intended isolation of independent aircraft operational functions residing in shared computing resources in all circumstances, including hardware and programming errors. The objective of Robust Partitioning is to provide the same level of functional isolation as a federated implementation (i.e., applications individually residing on separate computing elements). This means robust partitioning must support the cooperative coexistence of applications on a core processor, while assuring unauthorized, or unintended interference is prevented. » [12, p. 110]

Avant de continuer, mentionnons que cette description inclue la mention « on a core processor » (au singulier), ce qui implique qu'une définition formelle du cas à plusieurs coeurs n'est pas fournie dans cette norme.

Le partitionnement robuste, dans le contexte du modèle IMA, vise à émuler le modèle avionique fédéré dans un système avionique intégré. Ainsi, chaque fonction avionique s'exécutant sur un même processeur se verra isolée dans une « partition logicielle »[2] afin d'éviter que la défaillance d'une application n'en affecte une autre. Dans ce contexte, le mot « partition » est employé dans son sens informatique traditionnel : on sépare l'ensemble des ressources partagées d'un système en fragments isolés les uns des autres.

Rushby aborde de manière exhaustive le sujet du partitionnement dans les applications avioniques dans son rapport technique publié conjointement par la NASA et la FAA [57]. Il y présente intuitivement l'objectif-clé du partitionnement :

2. On réduira cette expression au seul mot « partition » pour simplifier le discours.

> « The behavior and performance of software in one partition must be unaffected by the software in other partitions. » [57, p. 11]

Sans partitionnement robuste, il serait impossible de partager une machine entre plusieurs fonctions avioniques de façon sécuritaire. La nécessité du partitionnement est confirmée par l'inclusion du concept dans DO-178B [1] et son document de clarification DO-248B [3].

L'implémentation du partitionnement robuste est réalisée par le « noyau » de partitionnement robuste. Dans notre cas, nous employons le mot « noyau » dans le sens informatique moderne : il s'agit de la composante centrale du système d'exploitation d'un ordinateur qui agit comme interface entre le matériel et les applications. Le bloc « Module OS » de la figure 2.2 représente le noyau de partitionnement robuste dans le modèle d'un système IMA.

On peut décomposer le partitionnement robuste en deux mécanismes interreliés dont la combinaison assure l'isolation désirée : le partitionnement temporel et le partitionnement spatial. Nous traiterons séparément de ces deux mécanismes tout au long de notre exposé.

2.3.2 Partitionnement spatial

Le partitionnement spatial vise à empêcher le partage de ressources physiques (mémoire, régions d'entrée/sortie) qui pourrait causer un couplage ou la transmission de défaillances entre les partitions. Chaque ressource partageable doit être assignée à une seule partition par une table de configuration. La table de configuration est statique, ce qui veut dire qu'elle est prédéfinie et qu'elle ne peut en aucun cas changer après le démarrage de la machine. On assigne aux partitions des zones de mémoire qui ne se chevauchent pas. Ensuite, la composante de partitionnement spatial du noyau de partitionnement robuste se charge de faire appliquer les restrictions de la configuration à l'aide de divers mécanismes matériels et logiciels. Nous discuterons plus en détail de ces mécanismes à la section 2.4. Grâce au partitionnement spatial, le dépassement de capacité d'un tampon mémoire dans une partition A ne pourrait en aucun

cas écraser de données dans une partition B. Pour le moment, on considérera le cas usuel d'une seule partition par application. Cette contrainte sera révisée au chapitre 4 lorsque nous présenterons notre modèle de partitionnement spatial multicoeur adapté.

2.3.3 Partitionnement temporel

Le partitionnement temporel sert quant à lui à garantir que chaque partition obtient l'usage exclusif du processeur pendant des périodes suffisantes pour respecter ses échéances critiques. On assigne des tranches de temps à chaque partition, en fonction de l'analyse du pire cas de temps d'exécution (« worst case execution time » en anglais, WCET) et des exigences de périodicité de chaque programme. Ces assignations sont combinées pour construire un plan d'exécution, défini sur la base d'un « bloc d'activation temporel majeur » périodique, dont la période est fixe. Le bloc d'activation temporel majeur est aussi connu sous le nom de « major time frame » en anglais. Les partitions obtiennent le contrôle du processeur durant des « fenêtres d'activation ». Ces fenêtres d'activations sont définies dans le bloc d'activation temporel majeur par leur durée et leur décalage temporel par rapport au début du bloc.

Un ordonnanceur applique ensuite le plan d'exécution préconfiguré. Encore ici, la configuration est fixe et ne peut pas changer après le démarrage du système. Contrairement à ce qui est fait dans les systèmes d'exploitation temps réel classiques, les ordonnanceurs employés pour le partitionnement temporel n'appliquent en aucun cas de préemption aux partitions. Ainsi, une partition qui aurait fini son traitement plus tôt que prévu n'accorderait pas le reste de sa tranche de temps à une autre partition. Tout au plus, le temps mort pourrait être réutilisé par le noyau pour des tâches administratives internes ou des fonctions ne pouvant pas mettre en péril la sûreté du système.

La figure 2.3 présente un exemple de plan d'exécution à deux partitions. Les paramètres de ce plan d'exécution sont résumés au tableau 2.1. Dans cet exemple, deux partitions se partagent le temps du bloc d'activation majeur. Les deux partitions obtiennent des fenêtres d'activation

totalisant la même durée totale, soit 12 ms. L'exécution de la partition A est séparée en deux fenêtres d'activations distinctes. On remarque que les fenêtres d'activation sont les mêmes dans chaque bloc d'activation majeur et qu'il est possible de laisser des fenêtres d'activation inoccupées (celles présentées en gris) pour permettre des extensions futures ou prendre en considération certains délais, tels que le changement de contexte.

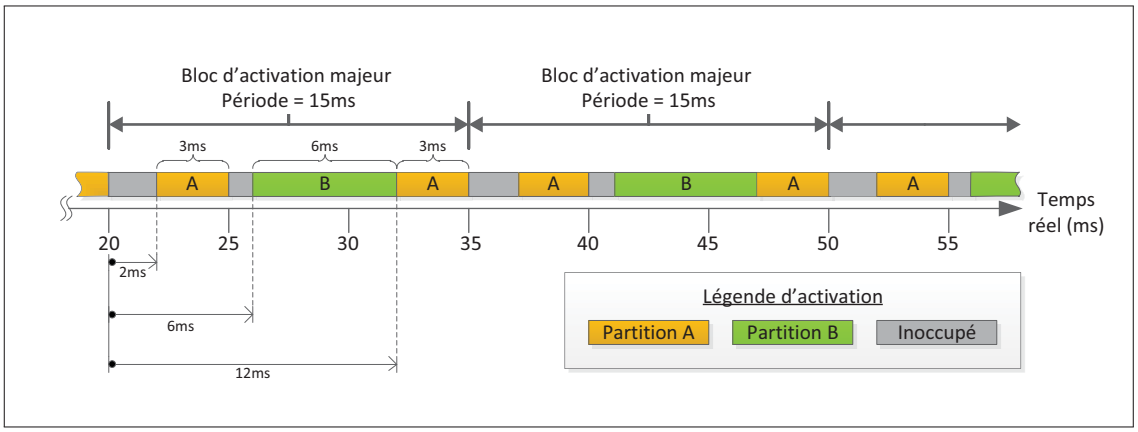


Figure 2.3 Exemple d'un plan d'exécution à deux partitions

Tableau 2.1 Paramètres de l'exemple d'un plan d'exécution à deux partitions de la figure 2.3

Fenêtre	Partition active	Décalage	Durée
1	A	2 ms	3 ms
2	B	6 ms	6 ms
3	A	12 ms	3 ms

2.3.4 Le partitionnement robuste selon la norme ARINC-653

L'augmentation de la popularité du modèle IMA dans les projets d'aviation militaire des années 90 a encouragé un mouvement de standardisation dans l'industrie avionique [49]. Ces efforts se sont soldés par la publication de la norme ARINC-653 [12] en 1997. Dans cette section, nous décrirons les principales caractéristiques d'ARINC-653 puisque son modèle de partitionnement est très commun en IMA.

La norme ARINC-653 définit un système exécutif d'applications (« APlication EXecutive », APEX) qui réalise le partitionnement robuste d'un processeur et de sa mémoire. De plus, elle définit une interface logicielle permettant la gestion des partitions, la communication entre ces dernières, ainsi que la gestion des conditions d'erreurs dans un système IMA. La norme est séparée en trois parties, mais nous ne faisons référence qu'à la partie 1 (ARINC-653P1, « Required Services »), qui spécifie l'APEX, les services de partitionnement essentiels et les mécanismes de configuration d'un système IMA.

Le modèle du système APEX est un noyau de partitionnement robuste où chaque partition apparaît comme une machine entièrement dédiée à l'application qui y est exécutée.

Outre le fait que la norme spécifie les caractéristiques spatiales d'une partition, aucune restriction n'est donnée quant à l'implémentation du partitionnement spatial sous ARINC-653. Les développeurs de systèmes d'exploitation sont libres d'employer les mécanismes d'isolation spatiale les mieux adaptés à l'architecture matérielle choisie.

Un système conforme à APEX emploie un ordonnanceur à deux niveaux pour le partitionnement temporel. Le niveau primaire est un ordonnanceur statique qui fournit des tranches de temps prédéfinies à chaque partition. Le niveau secondaire doit être un ordonnanceur préemptif à priorité fixe, équivalent à un système d'exploitation temps réel classique propre à chaque partition. Ce système d'exploitation de partition implémente aussi les services habituels d'un système d'exploitation temps réel.

La configuration d'un système ARINC-653 (appelée « System Blueprint » en anglais) est chargée à partir d'une table au démarrage et ne varie pas : elle est complètement statique. Il en va de même pour l'allocation de la mémoire, qui n'est permise que lors de la phase d'initialisation des partitions et du système d'exploitation. Le déterminisme est ainsi garanti et les opportunités de débordement causées par l'allocation de mémoire dynamique sont éliminées.

Un moniteur de la santé du système veille à surveiller les partitions. Si l'une d'entre elles venait à planter, une action de contingence serait exécutée et l'erreur serait communiquée aux partitions qui en dépendent. Il est possible que la partition soit redémarrée. Dans ce cas, il est spécifié que le redémarrage doit être complètement transparent pour les autres applications.

Dans un système d'exploitation ARINC-653, les partitions communiquent à travers des ports virtuels. Ces ports sont eux aussi prédéfinis dans la configuration du système. L'utilisation des ports virtuels permet l'abstraction du moyen de communication (local ou distant, ARINC-664 [11] ou autre) et le respect de l'isolation spatiale entre les partitions.

Depuis sa publication, ARINC-653 a été implémentée dans la majorité des systèmes d'exploitation avioniques commerciaux : Integrity-178B de Green Hills, LynxOS-178B de LynuxWorks, VxWorks 653 de WindRiver et PikeOS de SYSGO.

2.3.5 Interdépendances entre partitionnement spatial et temporel

Le partitionnement spatial et temporel sont interreliés à plusieurs niveaux. En particulier, l'implémentation du partitionnement spatial a toujours un effet sur le partitionnement temporel en raison des délais non nuls d'accès aux différents niveaux de la hiérarchie de mémoire sur les processeurs modernes.

Par exemple, la réalisation de la protection mémoire à l'aide d'une unité matérielle de gestion de la mémoire (« Memory Management Unit » ou MMU en anglais) peut causer des exceptions de fautes de pages. Ces exceptions sont nécessaires au bon fonctionnement des algorithmes de protection de mémoire, mais elles consomment du temps de traitement à la partition en cours.

Un autre exemple est l'effet du contenu des caches lors du changement de contexte de partitions. L'implémentation du partitionnement spatial doit gérer les caches afin d'empêcher une incertitude temporelle au niveau de l'exécution de la prochaine partition causée par le contenu de la cache de la partition précédente dans le plan d'exécution.

En raison de ces interdépendances fonctionnelles entre la composante spatiale et temporelle du partitionnement robuste, l'implémentation des noyaux de partitionnement est une combinaison d'une multitude de mécanismes dont la frontière est moins bien circonscrite qu'elle n'apparaît au premier coup d'oeil. Les noyaux de partitionnement robuste, malgré leur variété, font tous partie de la classe logicielle des systèmes d'exploitation. Ainsi, ils sont composés d'ordonnanceurs, de fonctionnalités de gestion de la mémoire, de pilotes de périphériques et d'autres services de base. Les différences se retrouvent dans les choix architecturaux effectués lors de la réalisation des composantes logicielles du noyau.

2.4 Méthodes de partitionnement spatial

Un scénario classique avant l'invention de la protection de mémoire était le plantage de toutes les applications d'un système en raison du débordement de mémoire d'une seule application. Sans mécanismes de protection de mémoire, un programme pouvait écrire partout en mémoire, incluant dans les zones déjà occupées par d'autres programmes ou par le système d'exploitation. Les premiers mécanismes de protection de la mémoire visaient donc à faire appliquer une politique de partitionnement spatial afin de permettre l'exécution sécuritaire de plusieurs programmes sur une même machine.

Au fil des développements dans le domaine des systèmes d'exploitation, l'apparition de nouveaux problèmes a motivé la conception de systèmes de protection de ressources partagées de plus en plus sophistiqués.

Le partitionnement spatial est réalisé avec une combinaison de méthodes logicielles et matérielles de protection des ressources partagées, couplées à un moniteur de santé du système qui applique des politiques de contingence lorsqu'une opération illégale survient.

Trois mécanismes matériels sont minimalement nécessaires pour permettre la protection des ressources partagées [52, p. 529] :

1. Le support d'au moins deux niveaux de privilèges matériels afin de pouvoir distinguer si le contexte d'exécution actuel est le noyau ou une application qui doit être restreinte. On parlera de mode « superviseur » et de « mode usager ». Dans certains cas, des « anneaux » concentriques représentent les niveaux de privilèges. Avec cette nomenclature les anneaux plus au centre sont plus privilégiés (plus près du coeur).
2. La protection des registres de contrôle de la machine contre l'écriture par les applications en mode usager. Des instructions réservées au mode superviseur doivent être employées pour modifier ces registres.
3. Le support d'une instruction permettant d'entrer en mode superviseur de manière contrôlée et une autre pour revenir en mode usager à partir du mode superviseur. La plupart des processeurs possèdent une version de l'instruction « system call » (appel système) qui permet de demander une opération privilégiée au logiciel superviseur. Cette instruction est toujours accompagnée d'une instruction « return from exception » (retour d'exception) qui quitte atomiquement le mode superviseur après l'exécution de l'opération privilégiée par le noyau.

Les mécanismes de gestion des privilèges sont employés en conjonction avec des mécanismes matériels de protection de la mémoire afin d'appliquer une politique de protection de la mémoire. Il existe deux types de ces mécanismes matériels de protection de mémoire : les unités de protection de mémoire (« Memory Protection Unit », MPU) et les unités de gestion de mémoire virtuelle (MMU). Dans les deux cas, le système d'exploitation configure les politiques d'accès dans ces unités et un accès interdit se solde par une exception.

Les MPU s'occupent uniquement de protection de la mémoire et se retrouvent habituellement dans les processeurs embarqués très simples ou lorsque la performance maximale est un critère.

Les MMU, quant à eux, gèrent la protection de la mémoire physique et permettent aussi d'employer un adressage à mémoire virtuelle. Ce type d'adressage permet la traduction entre des

adresses dites virtuelles, visibles au programmeur, et les adresses physiques de la mémoire, visibles uniquement au système d'exploitation. Plusieurs des mécanismes logiciels de partitionnement spatial sont basés sur la disponibilité d'un espace d'adresses virtuelles indépendant de l'espace d'adresses physiques [52]. Les processeurs de la famille PowerPC utilisés dans nos travaux possèdent tous un MMU flexible et performant.

Un système d'exploitation implémente la protection de la mémoire en combinant les mécanismes matériels de protection à des algorithmes de partitionnement de la mémoire. L'objectif est de configurer les mécanismes matériels afin de maximiser les performances de l'application, tout en proscrivant les accès interdits d'après la configuration du système. Souvent, le code exécutable d'un programme, ses données et sa pile seront placés dans des zones différentes de la mémoire, avec des permissions adaptées. Par exemple, la zone de données sera accessible en lecture et en écriture, alors que la zone de code ne le sera qu'en lecture-seule.

La granularité de la protection de mémoire est proportionnelle au niveau de sûreté désirée. Plus le niveau de sûreté est élevé, plus la granularité de protection sera fine afin de pouvoir déterminer précisément la cause d'un accès interdit. En connaissant précisément cette cause, il devient possible de réduire l'ampleur de l'action de contingence. Par exemple, l'accès interdit à un périphérique pourrait se solder par un simple signal d'erreur au lieu de la suspension de l'application.

Notons qu'il est possible de réaliser entièrement par logiciel la protection de mémoire nécessaire au partitionnement spatial. Cela est possible en insérant du code de vérification des bornes à chaque accès mémoire potentiellement dangereux [66]. Le code de vérification est inséré par le compilateur ou par le chargeur de programme. Cette approche est cependant sujette à la compromission lorsque le code de protection n'est pas géré par le noyau de partitionnement. De plus, la vérification des bornes engendre un coût en performances non-négligeable. On ne peut donc pas se fier aux méthodes entièrement logicielles dans le contexte IMA, à

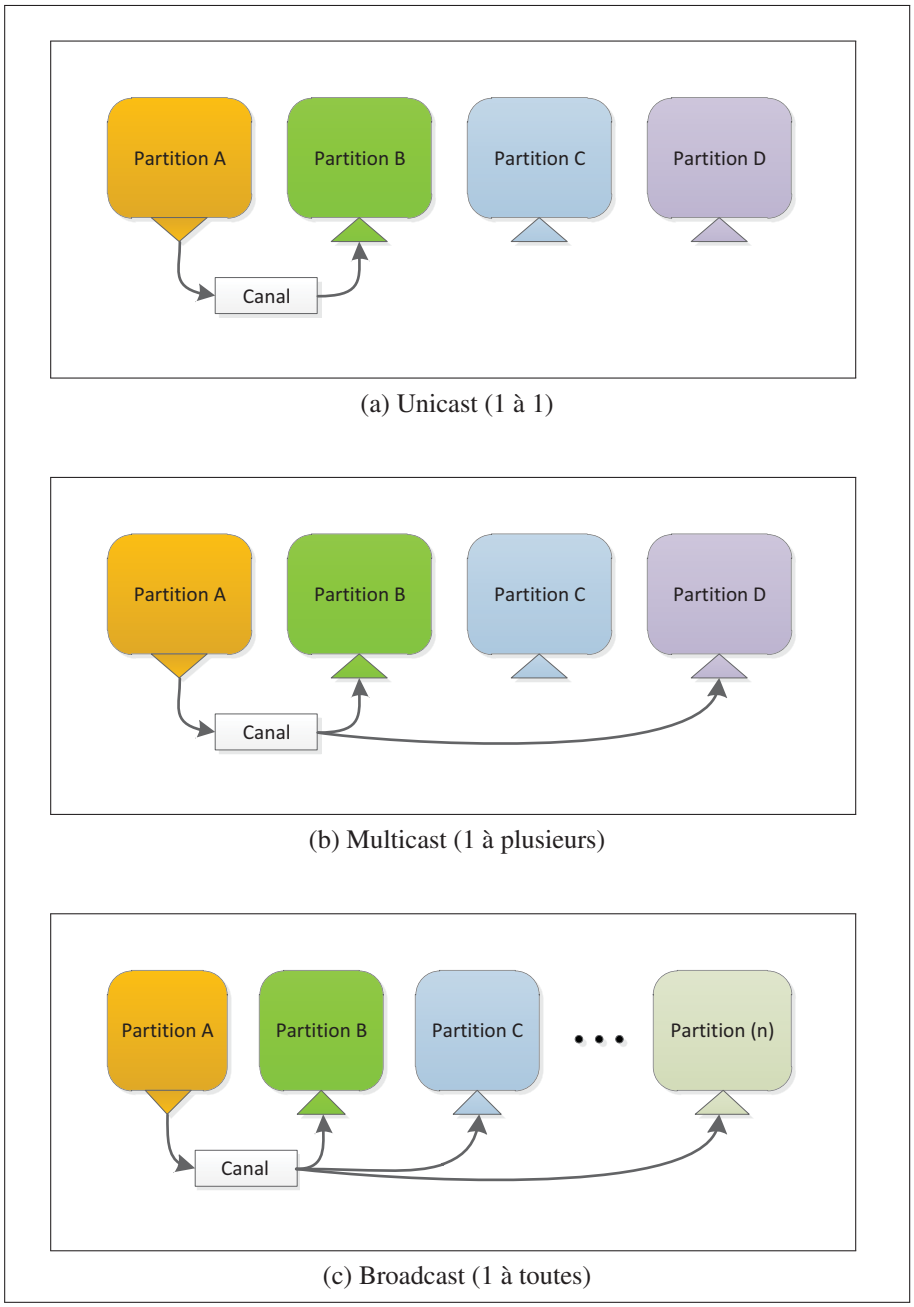
moins que toutes les applications soient programmées par des groupes coopératifs au sein de l'entreprise qui réalise l'intégration.

En plus des mécanismes de protection de la mémoire, l'implémentation du partitionnement spatial requiert des mécanismes de communication inter-partition (« inter-partition communication » en anglais, IPC). Ces mécanismes permettent aux applications d'interagir et de partager leurs résultats sans compromettre les barrières d'isolation. Le noyau de partitionnement robuste s'assure que toutes les communications sont autorisées selon des tables de privilèges préconfigurées. La plupart du temps, des restrictions s'appliquent quant à la taille et au nombre de messages permis entre chacune des partitions.

Considérons les mécanismes d'IPC définis dans la norme ARINC-653. Deux mécanismes y sont définis : les ports à échantillonnage (« sampling ports » en anglais) et les ports à file d'attente (« queuing ports » en anglais). On parle ici de « ports » de communication virtuels. Il ne s'agit pas de ports de communication physiques, mais plutôt d'abstractions, tels les ports d'entrée et de sortie d'une boîte noire. Comme leur nom l'indique, les ports à échantillonnage représentent l'abstraction d'un échantillonneur-bloqueur. À chaque fois qu'une nouvelle valeur y est inscrite par un producteur, elle demeure lisible sur le port par les consommateurs jusqu'à son expiration, le cas échéant. Inversement, les ports à file d'attente contiennent une file qui se vide au fur et à mesure que les données sont lues par les consommateurs.

L'interconnexion des ports de communication des différentes partitions est réalisée à travers des « canaux » virtuels. Dans ARINC-653, les canaux sont des liens logiques entre une source et une ou plusieurs destinations. Il est possible d'effectuer une assignation logique entre un port et une procédure ou un identifiant de pilote pour des réseaux de communication avionique, en plus d'une assignation vers des canaux inter-partition internes.

Dépendamment de l'implémentation, un même port de sortie peut être relié à plusieurs canaux permettant ainsi de réaliser une diffusion sélective. Les trois modes de diffusion définis dans ARINC-653 sont présentés à la figure 2.4.



(a) Unicast (1 à 1)

(b) Multicast (1 à plusieurs)

(c) Broadcast (1 à toutes)

Figure 2.4 Modes de diffusion de messages inter-partitions

2.5 Méthodes de partitionnement temporel

Alors que le partitionnement spatial vise à prévenir les partages de ressources indésirables, le partitionnement temporel vise à garantir des ressources de traitement suffisantes aux partitions.

L'implémentation du partitionnement temporel dans un noyau de partitionnement robuste se résume à la combinaison d'un ordonnanceur de partitions et d'un ensemble de règles qui enrayent la subtilisation de cycles d'une partition par une autre.

L'ordonnanceur statique cyclique est le plus populaire en partitionnement temporel robuste en raison de sa simplicité. Le seul critère de changement de contexte de ce type d'ordonnanceur est l'avancement du temps. Son implémentation doit être extrêmement robuste et sûre, car l'intégrité du partitionnement temporel en dépend directement. Comme nous l'avons déjà mentionné, la norme ARINC-653 mandate l'utilisation de ce type d'ordonnanceur comme premier niveau d'ordonnancement dans l'APEX.

En plus de l'ordonnanceur, la gestion des interruptions est un élément clé afin d'éviter les dépassements temporels des partitions. En particulier, la gestion des interruptions du système doit être centralisée dans le noyau. Les partitions ne peuvent avoir accès qu'à des interruptions virtuelles et les sources d'interruptions doivent être rigoureusement contrôlées afin d'éviter qu'une source imprévue n'interrompe la partition en cours, même pour un très court laps de temps.

Finalement, les autres fonctions avancées des processeurs, tels que l'accès direct à la mémoire (« Direct Memory Access » ou DMA, en anglais) et les bus partagés d'accès aux périphériques, doivent aussi être gérées par le noyau de partitionnement robuste. Par exemple, les modules DMA peuvent subtiliser des cycles d'accès au bus de mémoire si leur gestion est complètement indépendante des réalités du plan d'exécution. D'ailleurs, certains noyaux de partitionnement robuste simples, tels que POK [22] et XtratuM [19] n'offrent aucune fonctionnalité intégrée pour l'exploitation des modules DMA.

2.6 Choix d'un noyau de partitionnement robuste à adapter

Lors de la phase d'exploration de nos travaux, nous avons cherché des noyaux de partitionnement robuste académiques existants afin d'en trouver un qui pourrait servir de base à notre adaptation multicoeur. Nous pouvons classifier les noyaux identifiés au sein de trois catégories : 1) ceux qui supportent directement ARINC-653, 2) ceux qui supportent des modèles similaires à ARINC-653 et 3) ceux qui ne supportent pas des modèles similaires à ARINC-653.

Parmi ceux qui supportent directement ARINC-653, nous retrouvons les projets AIR et POK. AIR est un projet de collaboration industrielle entre l'Université de Lisbonne et Skysoft Portugal, supporté par l'agence spatiale européenne (« European Space Agency » ou ESA, en anglais). Le nom AIR provient de l'acronyme « ARINC-653 In RTEMS ». Comme son nom l'indique, le projet AIR visait à adapter le système d'exploitation temps réel à code source libre RTEMS, déjà employé dans le domaine spatial, afin de supporter un modèle de partitionnement ARINC-653. La première phase du projet (AIR1 [56]), achevée en novembre 2007, visait à déterminer les adaptations nécessaires à RTEMS pour supporter l'APEX. La deuxième phase du projet (AIR2 [58]), achevée en novembre 2009, visait la réalisation d'une preuve de concept d'une implémentation complète de l'APEX par-dessus RTEMS sous les architectures IA32 et LEON. Le noyau de partitionnement robuste POK [22], quant à lui, est un projet démarré par Julien Delange à TELECOM ParisTech. Le nom POK provient de l'acronyme « Partitioned Operation Kernel ». POK est un noyau temps réel conçu dès le départ pour les applications partitionnées. En particulier, les fonctions critiques à la sûreté sont réalisées dans un noyau temps réel minimal, alors que les différentes interfaces de partitionnement robuste, dont l'APEX, sont réalisées à travers des librairies qui font appel aux services de base du noyau. La conception minimaliste du noyau simplifie sa vérification par des outils d'analyse statique et de couverture de code. POK vise principalement l'utilisation en recherche sur le

partitionnement aidé par des outils de modélisation de systèmes. Les architectures supportées par POK sont LEON, PowerPC et IA32.

Dans la catégorie des noyaux qui supportent un modèle similaire à ARINC-653, on retrouve le noyau XtratuM [19]. Ce projet est une collaboration entre l'Université Polytechnique de Valence, Astrium SAS et Teletel. Comme AIR, ce projet est supporté par l'ESA et vise l'application du partitionnement robuste au domaine spatial. La cible de ce noyau est LEON. XtratuM est un « hyperviseur » de type 1 selon la classification de Goldberg [31], soit un moniteur de machines virtuelles natives. Contrairement à AIR et POK, XtratuM réalise le partitionnement à l'échelle de machines virtuelles complètes, au lieu de gérer un ensemble de tâches dans le noyau. Il est ainsi possible d'implémenter des systèmes d'exploitation de partition qui ont l'illusion d'avoir le contrôle total de la machine. Cependant, XtratuM ne supporte pas directement l'APEX, ni la description de configuration spécifiée dans ARINC-653. Au lieu de cela, un modèle de partitionnement et un ensemble de fonctions calqués sur la sémantique ARINC-653 sont fournis. Il serait possible de réaliser un système compatible ARINC-653 basé sur XtratuM en programmant un système d'exploitation de partition qui emploierait les fonctionnalités du noyau XtratuM pour réaliser les interfaces logicielles spécifiées dans la norme.

Nous n'avons pas étudié en détail les noyaux temps réel dont les objectifs de conception n'étaient pas en lien avec le domaine aérospatial. Nous mentionnerons par contre les micronoyaux basés sur le modèle L4 [40], qui peuvent servir à construire des systèmes partitionnés. Parmi la famille L4, les plus intéressants sont L4Ka::Pistachio [39] et OKL4 [50], car ces deux noyaux supportent les processeurs multicoeurs et servent à partitionner des systèmes embarqués. Dans la famille L4, on retrouve aussi seL4 [38] (« Secure Embedded L4 »), une implémentation dont il a été formellement prouvé qu'elle respecte ses spécifications d'origine.

Lorsqu'est venu le moment de choisir un point de départ pour notre adaptation, nous avons considéré les noyaux XtratuM et AIR. Ces deux noyaux offraient un modèle de partitionne-

ment basé sur ARINC-653 et étaient dotés d'une suite d'outils et d'exemples complets. Nous avons appris l'existence du support ARINC-653 dans POK plusieurs mois après le début de nos travaux, c'est pourquoi ne l'avons pas considéré.

Dès le départ, AIR semblait plus adéquat, car son état d'avancement était meilleur que celui de XtratuM. En effet, à ce moment, XtratuM en était à la version 2.1, qui était instable et ne possédait qu'un support minimal des fonctions de communication inter-partitions. Après avoir contacté les responsables du projet AIR, il s'avérait que le code source de ce noyau n'était pas disponible en dehors de leur projet. Il ne nous restait donc comme alternative que XtratuM. XtratuM était plus attrayant qu'AIR au niveau de la simplicité, car le support de RTEMS nécessaire à AIR était lourd et complexe. Finalement, après avoir contacté les auteurs de XtratuM, nous avons réussi à obtenir le code source en licence GPL (« GNU Public License ») de XtratuM 2.2.2, une version beaucoup plus avancée que la version 2.1 évaluée initialement. En raison de la durée restante au projet, nous avons choisi de commencer immédiatement l'adaptation de XtratuM au lieu de rechercher plus en profondeur des alternatives. Malgré certains désavantages, dont un manque de documentation dans le code source, XtratuM s'est avéré être un bon choix pour nos travaux. En effet, la simplicité de XtratuM et son niveau d'achèvement nous ont permis d'être immédiatement productifs.

2.7 Processeurs multicoeurs

Nos travaux visent l'adaptation d'un noyau de partitionnement robuste monocoeur vers une architecture multicoeur. Les processeurs multicoeurs modernes sont des descendants directs des modèles d'ordinateurs à multiples processeurs discrets réalisés dans les années 80 et 90. Le type de processeurs multicoeurs qui nous intéresse contient plusieurs coeurs séquentiels identiques, reliés à un module de gestion d'accès vers la mémoire de masse partagée. Les différents types d'architectures multicoeurs modernes et les modèles de programmation associés sont couverts en détail dans un ouvrage récent de Solihin [59].

La principale différence entre l'utilisation des processeurs monocoeurs et celle des processeurs multicoeurs est le besoin d'exploiter le parallélisme explicitement lors de la programmation. Malheureusement, il n'existe actuellement pas de processeurs qui distribuent automatiquement le traitement entre les coeurs. Les programmeurs doivent donc modifier leurs programmes afin d'exploiter le parallélisme exposé par la plateforme. Nous recommandons l'ouvrage de Herlihy et Shavit [32] pour une couverture complète des méthodes de programmation pour l'exploitation du parallélisme exposé par les processeurs multicoeurs.

En plus d'employer des architectures différentes, les programmes qui exploitent le parallélisme de plusieurs coeurs doivent synchroniser leurs opérations à certains moments. La synchronisation est nécessaire pour éviter les conditions de courses et les bogues qui peuvent survenir lorsque des évènements importants ne sont pas séquencés adéquatement. La synchronisation dans les systèmes multiprocesseurs est un problème suffisamment important pour qu'un sous-domaine des sciences de l'informatique s'y rattache spécifiquement.

Tous les manuels de niveau universitaire au sujet des systèmes d'exploitation contiennent un traitement des problèmes classiques de synchronisation en général. L'article séminal de Mellor-Crummey et Scott [48] est une bonne introduction aux primitives de synchronisation multiprocesseurs pour les systèmes à mémoire partagée. Nous recommandons aussi les livres de Solihin [59] et de Herlihy et Shavit [32] pour un traitement approfondi de la synchronisation et des structures de données parallèles sur les processeurs multicoeurs récents.

Les autres notions d'architecture des processeurs multicoeurs seront introduites au fur et à mesure des prochains chapitres, aux moments opportuns.

CHAPITRE 3

ARCHITECTURE ET IMPLÉMENTATION DU NOYAU XTRATUM SUR PROCESSEURS LEON

3.1 Survol

Ce chapitre couvre l'architecture et l'implémentation du noyau de partitionnement robuste XtratuM sur les processeurs LEON. Avant d'entamer l'analyse de l'adaptation multicoeur de XtratuM, il est important de présenter les différents blocs qui le composent. Cela est d'autant plus important que ces détails ne sont pas disponibles dans des rapports publics. En effet, les articles de conférence à propos de XtratuM [19, 20] et son manuel de l'utilisateur [46] ne fournissent qu'un survol de haut niveau. Pour cette raison, nous avons effectué l'ingénierie inverse du noyau à partir des sources afin de pouvoir déterminer quelles seraient les adaptations architecturales nécessaires. Le résultat de cette ingénierie inverse est le présent chapitre.

Lors du processus d'analyse, nous avons décomposé chacun des blocs fonctionnels du noyau. Nous présentons dans les prochaines sections l'implémentation originale sur processeurs LEON de chacun d'entre eux. Ces détails permettent de mieux comprendre les aspects pragmatiques de l'implémentation d'un noyau de partitionnement robuste complet. Au chapitre 4, nous suivrons un ordre identique des blocs fonctionnels lorsque nous présenterons l'adaptation multicoeur de l'architecture du noyau.

3.2 Historique de XtratuM

L'objectif initial de XtratuM 1.0 était de permettre la cohabitation de systèmes d'exploitation temps réel sur une même machine dont le système d'exploitation principal était Linux [45]. Dans cette configuration, XtratuM était déployé en module de noyau Linux. Les interruptions étaient redirigées après le démarrage de Linux afin de donner le contrôle à XtratuM comme

micronoyau maître. Le micronoyau pouvait ensuite assigner des partitions temporelles et spatiales à des systèmes d'exploitation invités tels que RTLinux/GPL, MarteOS, etc. À la suite des travaux de la série 1.0, l'équipe de XtratuM s'est redirigée vers la réalisation d'un hyperviseur temps réel pour applications spatiales (c.-à-d. satellites). L'objectif de ces nouveaux développements était d'éliminer la dépendance envers le noyau Linux et d'ajouter des mécanismes de partitionnement robuste similaires à ceux des systèmes d'exploitation ARINC-653 [19]. Le nom XtratuM est demeuré, malgré le fait que le code soit complètement réécrit et les spécifications complètement nouvelles. Le numéro de version majeure a cependant été changé à 2.0 pour refléter l'évolution du noyau.

Dans le reste du chapitre, nous évaluerons exclusivement la série 2.0 de XtratuM. Ainsi, lorsque nous emploierons le nom XtratuM, cela fera implicitement référence à l'architecture des versions $\geq$2.0.

3.3 Architecture de XtratuM

Le noyau XtratuM a été conçu pour la famille de processeurs LEON de la compagnie Aeroflex Gaisler. Cette famille est basée sur l'architecture RISC SPARCv8 [60] et conçue spécifiquement pour les applications spatiales. Les processeurs LEON intègrent, en plus d'un coeur SPARCv8, tous les périphériques et interfaces nécessaires à la réalisation d'un système embarqué de charge utile de satellite.

L'architecture de XtratuM est celle d'un moniteur de machine virtuelle, ou hyperviseur. Ainsi, le noyau XtratuM gère les ressources matérielles partagées de la machine afin de fournir l'illusion de plusieurs machines virtuelles indépendantes. Les partitions sont donc réalisées sous la forme de machines virtuelles, contrairement à l'approche de certains OS ARINC-653 qui réalisent le partitionnement en isolant des groupes de tâches à l'aide de restrictions additionnelles dans le système d'exploitation. Dans le contexte de XtratuM, les mots partition et machine virtuelle sont synonymes. En effet, dans le contexte IMA, la partition est l'unité descriptive d'un

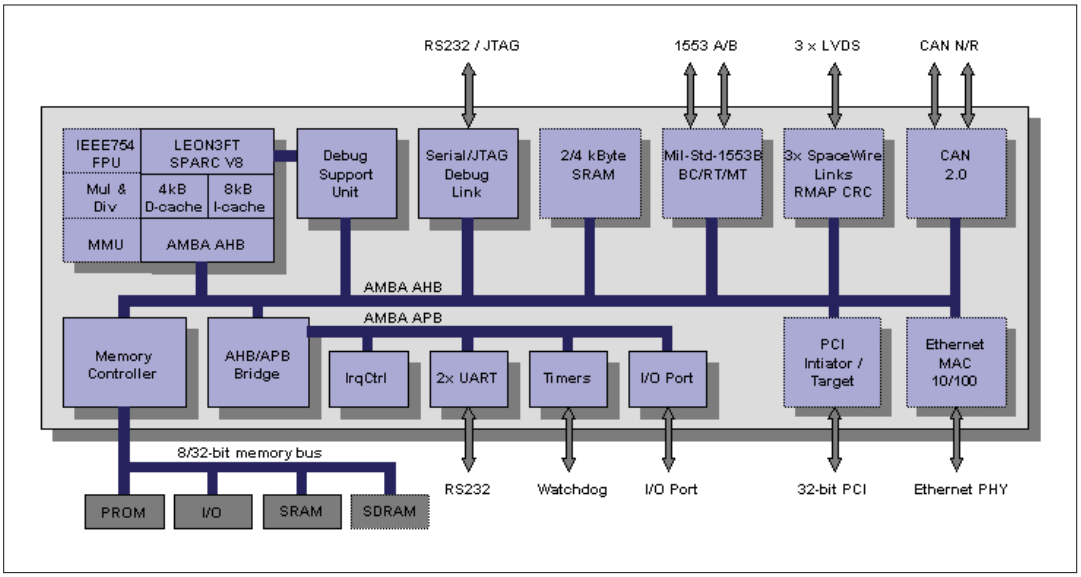


Figure 3.1 Schéma-bloc typique d'un processeur LEON3, tirée de [7] avec la permission d'Aeroflex Gaisler. ©2008 Aeroflex Gaisler AB

ensemble de ressources matérielles appartenant à une application indépendante. De même, dans XtratuM, les machines virtuelles sont dotées d'un ensemble de ressources réservées à une application indépendante, telle que des régions de mémoire, des périphériques et des sources d'interruptions. Nous utiliserons donc l'appellation « partition » au lieu de « machine virtuelle » afin de demeurer en harmonie avec les publications du domaine de l'IMA, étant donné l'équivalence des concepts dans notre cas.

XtratuM est un hyperviseur de type 1 selon la nomenclature classique de Goldberg [31]. Les hyperviseurs de type 1 sont natifs, c'est-à-dire qu'ils prennent le contrôle total de la machine. Les hyperviseurs de type 2, quant à eux, sont supportés par un OS et emploient les services de l'OS hôte pour réaliser la virtualisation.

Chaque partition est gérée de manière à assurer une isolation spatiale et temporelle robuste par rapport à l'ensemble des autres partitions du système. Les partitions sont toutes exécutées en mode usager non privilégié. Elles ne peuvent donc en aucun cas prendre le contrôle des

ressources critiques du système sans causer d'exception de privilège. Seul le noyau est exécuté en mode superviseur privilégié.

Le noyau permet aux partitions d'avoir accès à des ressources matérielles et à des services qui sont autorisés dans le plan de configuration du système (PCS). Toutes les transactions de services doivent s'effectuer à travers des appels au noyau hyperviseur afin qu'il soit possible de valider les droits d'accès configurés dans le PCS. Les appels au noyau sont réalisés à l'aide du mécanisme d'appel système (« system call ») du processeur.

Après le démarrage du noyau, un ordonnanceur cyclique détermine quelles partitions sont activées à quels moments, en fonction d'un plan d'exécution prédéterminé dans le PCS. Les partitions peuvent se retrouver dans un de quatre modes de fonctionnement (ou états). Le diagramme des états possibles est présenté à la figure 3.2. Dans ce diagramme, les arcs représentent des appels systèmes réalisés par le noyau ou par une partition « superviseur » en cas de détection d'erreur. Les modes de fonctionnement possibles sont les suivants :

- *Boot :* Démarrage et préparation de la machine virtuelle, incluant l'initialisation des registres de contrôle virtuels et démarrage d'un OS de partition. La partition est activée dans sa fenêtre d'activation du plan d'exécution.
- *Normal :* La partition est en fonctionnement normal. Elle est activée dans sa fenêtre d'activation du plan d'exécution.
- *Suspend :* La partition est suspendue. Elle n'est pas activée dans sa fenêtre d'activation et les interruptions qui lui appartiennent ne sont pas traitées. Cet état peut être atteint si une faute est détectée, afin d'empêcher l'exécution continue dans un état erroné. Il est possible de faire revenir la partition en mode normal.
- *Halt :* La partition n'est plus jamais activée et il n'est pas possible de la faire revenir en mode normal sans effectuer un redémarrage (mode « Boot »). La fenêtre d'activation as-

signée à la partition demeure inutilisée et toutes les ressources appartenant à la partition sont relâchées.

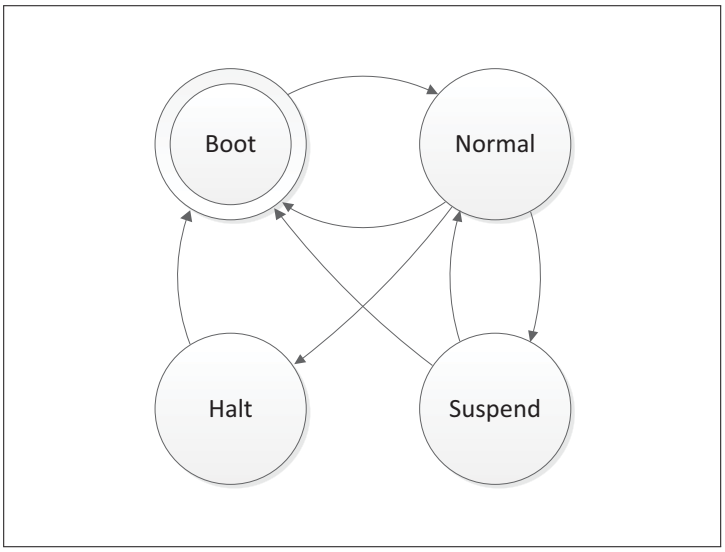


Figure 3.2 Diagramme des états possibles
pour une partition dans XtratuM

Il existe deux types de partitions : le type « superviseur » et le type « normal ». Les partitions de type superviseur ont accès à certains appels du noyau qui permettent la récupération des partitions en cas d'erreur et la gestion du moniteur de santé du système. Par exemple, une partition superviseur se voit en droit de redémarrer n'importe quelle partition du système alors qu'une partition normale ne pourrait redémarrer qu'elle-même. Dans le cas habituel, des partitions sont assignées à une fonction de gestion des défaillances (« Integrated Vehicle Health Management » ou IVHM) afin de déporter ce type de logique en dehors du noyau à haut niveau de sûreté. Ces partitions IVHM sont de type superviseur afin d'avoir suffisamment de privilèges pour contenir les défaillances et essayer des stratégies de recouvrement. À l'opposé, les partitions normales n'ont pas de privilège de gestion afin de limiter les cas de défaillances qui pourraient se propager en réalisant, par exemple, des redémarrages inopinés d'autres partitions.

3.4 Plan de configuration du système

Le plan de configuration du système (PCS) est une structure de donnée hiérarchique qui décrit statiquement tous les aspects du système. Le modèle de PCS de XtratuM est très proche du modèle de base spécifié dans la section 5.0 d'ARINC-653. Le PCS décrit la configuration des aspects suivants du système :

- *Partitionnement spatial :* les zones de mémoire et d'entrées/sorties associées à toutes les partitions et au noyau.
- *Partitionnement temporel :* le plan d'exécution du système et les contraintes temporelles des différentes partitions ;
- *Communication inter-partitions :* les connexions de ports virtuels qui permettent la communication entre les partitions ;
- *Interruptions :* l'assignation des lignes d'interruptions matérielles à chaque partition ;
- *Moniteur de santé du système :* l'action à prendre pour chacune des exceptions et défaillances détectables dans chaque partition ;
- *Pilotes du noyau :* les paramètres de configuration de chacun des pilotes disponibles dans le noyau.

Le PCS de XtratuM est décrit dans un fichier structuré XML et validé par un schéma en langage *W3C XML Schema Language* [70]. L'intégrateur fournit aux développeurs de partitions le PCS du système final afin de fixer leurs limites en ressources. Il n'y a qu'un seul PCS par système déployé et ce dernier décrit toutes les partitions.

Lors de la compilation de l'image du système et des partitions, le PCS est employé pour générer des tables de configuration et des fichiers intermédiaires. La validation du PCS et la compilation de la table de configuration binaire sont réalisées à l'aide de l'outil `xmcparser` fourni avec XtratuM.

La validation du PCS s'effectue en deux phases. Une première phase valide le fichier XML selon le schéma XML du PCS. Cette phase détecte les éléments manquants, les erreurs de syntaxe et les valeurs hors limites sans prendre en considération la sémantique des données. La deuxième phase analyse les structures de données afin de détecter des erreurs logiques dans la description du système. Des erreurs telles que le chevauchement spatial de partitions ou le débordement temporel du plan d'exécution sont détectées.

Après la validation, le PCS est compilé en table de configuration binaire compatible avec les structures internes du noyau. Le flot de compilation de la table de configuration est présenté à la figure 3.3. Notons que `xmcparser` génère la table de configuration sous la forme d'un fichier de code source C. Cela permet d'utiliser le compilateur de la plateforme cible pour générer une table binaire directement compatible avec la représentation mémoire des structures internes du noyau. Cependant, le fichier de code C de la table de configuration n'est pas lié avec le reste du code du noyau, mais plutôt compilé séparément et inséré comme un fichier intégré dans l'image du système.

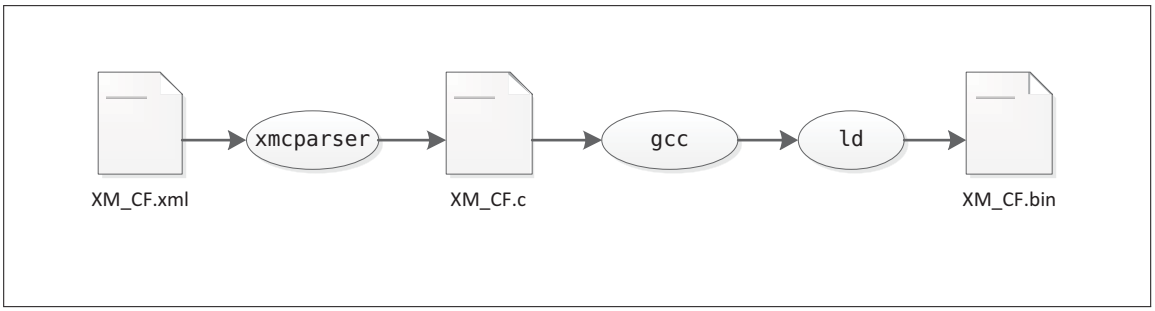


Figure 3.3 Flot de compilation de la table de configuration de XtratuM

Lors de la phase de compilation de chacune des partitions, une recherche dans le fichier XML du PCS est utilisée pour extraire les adresses de départ des binaires pour l'édition de liens statique. À partir de cette information, le script d'édition de liens est généré pour forcer l'assignation de cette adresse statique au début du binaire compilé de la partition.

3.5 Déploiement du système partitionné

Le système est déployé à partir d'une image qui regroupe les binaires du chargeur de programme (« bootloader »), du noyau, de la table de configuration du PCS et des partitions. Cette image peut directement être chargée en ROM sur une carte de développement LEON.

Le chargeur de programme est nommé « resident software » (RSW) dans la distribution XtratuM. Ce chargeur place en mémoire tous les binaires aux endroits prédéfinis dans la table de configuration du PCS et démarre ensuite le noyau.

Après le démarrage du noyau, le système est entièrement reconfiguré par des routines d'initialisation et des pilotes matériels intégrés à XtratuM. Il n'y a plus aucune dépendance envers quelconque micrologiciel préinstallé sur le matériel par le manufacturier. La gestion de toutes les interruptions et de tous les périphériques se fait à partir de XtratuM, ce qui permet d'éviter toute défaillance causée par des éléments logiciels externes au noyau de partitionnement robuste.

Les sous-sections suivantes présentent les cinq étapes nécessaires au déploiement d'une image du système avec XtratuM.

3.5.1 Étape 1 : Configuration et construction du noyau

Dans cette étape, le noyau est configuré, puis construit. La configuration est réalisée à l'aide d'un script muni d'une interface graphique qui permet de spécifier les valeurs de tous les paramètres critiques du noyau. Par exemple, il est possible de spécifier le nombre maximal de partitions supportées, les zones mémoire assignées au noyau, le type de processeur cible, etc. Le script de configuration génère des fichiers d'en-tête pour l'inclusion dans le code source C, ainsi qu'un fichier de configuration pour l'outil de construction `make`. Les fichiers d'en-tête contiennent des constantes qui permettent la compilation conditionnelle du noyau en fonction

de la configuration spécifiée. La figure 3.4 présente une capture d'écran d'une page affichée par le script de configuration.

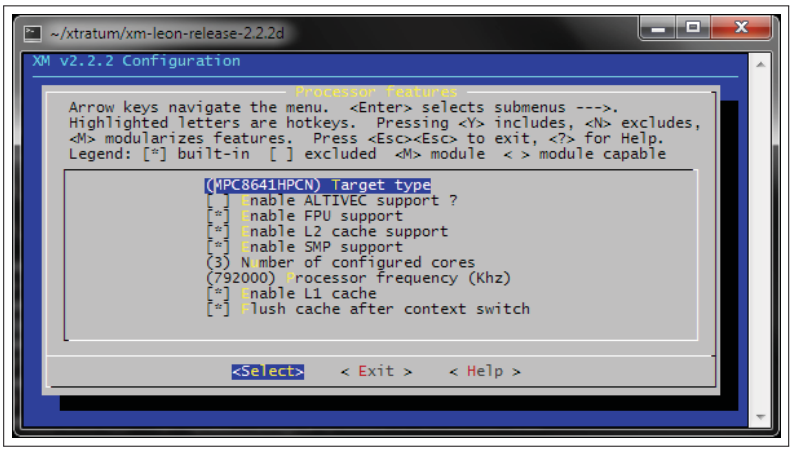


Figure 3.4 Fenêtre du script de configuration du noyau XtratuM

Après la configuration, le noyau est construit à l'aide de l'outil standard `make`. Le noyau complet est compilé en deux versions : une image binaire à charger nommée `xm_core.bin` et une image ELF nommée `xm_core` permettant le débogage. Le noyau, comme tous les autres éléments d'un système basé sur XtratuM, est lié statiquement.

Plusieurs autres éléments sont construits en même temps que le noyau en tant que tel :

- la librairie de services `libxm.a`, qui fournit les routines d'accès aux services du noyau et qui doit être liée avec chaque partition ;
- l'outil de validation et de compilation du PCS, nommé `xmcparser` ;
- l'outil de construction de l'image du système, nommé `xmpack` ;
- le schéma XML du PCS, nommé `xmc.xsd`, en version prête à l'emploi avec ses paramètres remplis selon les choix effectués au moment de la configuration.

Après la construction du noyau, il est toujours nécessaire de reconstruire les partitions et les images systèmes, car ces livrables dépendent de fichiers générés lors de la construction du noyau.

3.5.2 Étape 2 : Génération de la table de configuration binaire à partir du PCS

Après la construction du noyau, on génère la table de configuration binaire qui lui est associée, en fonction de la configuration globale du système dans le PCS. Cette étape est décrite à la section 3.4. La table de configuration binaire résultante se trouve dans un fichier nommé `xm_cf.bin`.

3.5.3 Étape 3 : Construction des partitions

Lorsque le noyau et la table de configuration binaire du PCS sont construits, il devient enfin possible de construire les binaires de partitions. Chaque partition est construite selon les besoins des développeurs d'applications. Le flot de construction à suivre n'est pas spécifié. Il est par contre nécessaire que chaque partition construite respecte les conventions suivantes :

- la librairie de services `libxm.a` doit être liée avec le binaire afin de pouvoir accéder aux services offerts par le noyau ;
- le binaire final doit être assemblé pour un démarrage à l'adresse spécifiée dans le PCS ;
- la partition doit au minimum appeler les services d'initialisation de `libxm.a`.

3.5.4 Étape 4 : Construction de l'image de déploiement du système

L'image de déploiement regroupe le chargeur RSW, le noyau, la table de configuration binaire et les binaires de partitions. Elle est construite à l'aide de l'outil `xmpack`. Cet outil regroupe tous les fichiers en une seule archive. Il génère ensuite une en-tête qui permet au chargeur RSW de placer toutes les images dans les bonnes zones de mémoire lors du démarrage. Le format de fichier de l'image de déploiement est décrit en détail dans le manuel de l'usager de XtratuM [46].

L'image finale, prête à l'écriture en ROM, est nommée `container.bin`. Avant la génération de container.bin, une image ELF du chargeur RSW est créée, ce qui permet de déboguer le chargeur RSW si nécessaire.

3.5.5 Étape 5 : Déploiement de l'image sur la carte matérielle

La dernière étape du déploiement est le transfert vers la carte matérielle. L'image du système est écrite dans la ROM de la carte de développement ou du système final et XtratuM peut démarrer. Lors du prototypage, il est aussi possible de charger soit l'image binaire, soit l'image en format ELF dans une plateforme virtuelle. La carte matérielle n'est alors pas nécessaire.

3.6 Services de base du noyau

Comme nous l'avons déjà mentionné, XtratuM est un hyperviseur et non pas un système d'exploitation temps réel classique. Une différence majeure entre ces deux technologies se situe au niveau de la méthode d'isolation des fonctions indépendantes.

Les systèmes d'exploitation temps réel classiques regroupent les fonctions indépendantes en groupes de tâches et les isolent en employant les mécanismes déjà présentés à la section 2.4. Toutes les tâches se trouvent cependant gérées par le même noyau.

À l'opposé, les hyperviseurs ne font aucune gestion directe des tâches. Au lieu de cela, chaque fonction se voit assigner une machine virtuelle autonome. Chaque fonction est donc exécutée, à peu de choses près, comme si une machine physique indépendante lui appartenait.

La virtualisation de la machine est réalisée en « enfermant » chaque partition dans une tâche s'exécutant en mode usager. L'hyperviseur s'occupe d'effectuer le changement de contexte des partitions selon le plan d'exécution.

Le noyau XtratuM fournit une série de services aux partitions. Certains services leur permettent de gérer l'environnement virtuel dans lequel elles se trouvent. D'autres services sont liés à la communication inter-partitions et au moniteur de santé du système. Tous ces services sont appelés à travers une interface centralisée qui passe entre la barrière d'isolation et le

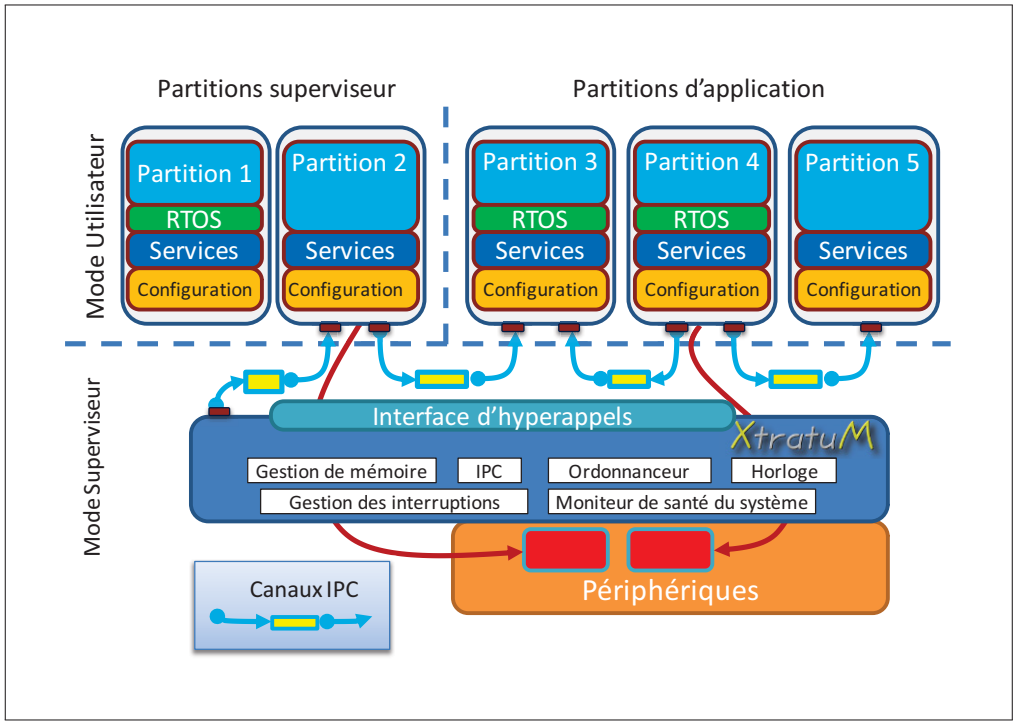


Figure 3.5 Schéma de l'architecture de XtratuM incluant les services

noyau. Le mécanisme d'hyperappels est la réalisation concrète de cette interface centralisée. La section 3.7 traite des détails du mécanisme d'hyperappels dans XtratuM.

Les services de base du noyau XtratuM sont décrits dans les sous-sections suivantes. Les autres blocs d'importance du noyau sont couverts dans les sections 3.9 à 3.11.

3.6.1 Gestion de l'horloge

Le service de gestion de l'horloge est primordial dans tous les systèmes d'exploitation et hyperviseurs, particulièrement dans le cas des systèmes temps réel où les contraintes temporelles requièrent une gestion précise du temps.

Le service de gestion de l'horloge de XtratuM fournit une série de minuteries et d'horloges. Les minuteries génèrent des évènements après une durée prédéterminée. Les horloges fournissent l'heure du système sur 64 bits. La résolution temporelle des horloges et minuteries de XtratuM est d'une microseconde.

Le noyau fournit une vue réelle et une vue virtuelle du temps. Le temps réel est basé sur le temps système depuis le démarrage et il est absolu. Le temps virtuel, quant à lui, n'est seulement comptabilisé que durant la fenêtre d'activation d'une partition. Le temps virtuel permet aux partitions d'ordonnancer des évènements dans une ligne du temps relative à une partition. Par exemple, cela permettrait d'assurer le fonctionnement correct d'un ordonnanceur de tâches tourniquet (de l'anglais « round-robin ») à l'intérieur d'une partition. Ainsi, si le contexte de tâche doit être changé toutes les 20 millisecondes, cela sera possible sans aberrations temporelles, peu importe combien de fois la fenêtre d'activation de la partition s'est arrêtée durant cette période. La figure 3.6 illustre la différence de progression entre le temps réel et le temps virtuel avec un exemple synthétique à deux partitions. On remarque dans cette figure que le temps virtuel de la partition « A » est complètement arrêté pendant l'exécution d'une autre partition. Si la partition « A » ne se fiait qu'au temps virtuel, il lui serait même impossible de détecter que d'autres partitions sont en fonction dans le système.

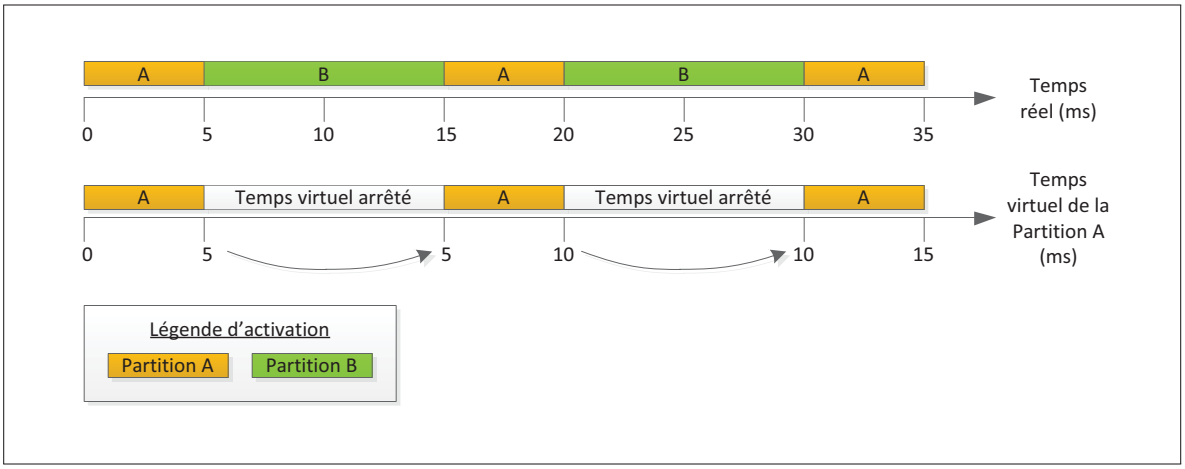


Figure 3.6 Illustration de la progression du temps réel et du temps virtuel dans XtratuM

La version LEON de XtratuM réalise les horloges et minuteries avec une combinaison de logiciel et de matériel. Deux minuteries matérielles sont utilisées : une pour l'horloge principale du noyau et une pour la minuterie principale du noyau.

Pour résumer, chaque partition a accès aux services temporels suivants :

- l'horloge globale en temps réel ;
- une horloge virtuelle locale ;
- une minuterie basée sur le temps réel ;
- une minuterie basée sur le temps virtuel ;
- une minuterie chien de garde (« watchdog ») basée sur le temps virtuel dont le fonctionnement est expérimental dans XtratuM 2.2.2.

Il est possible pour les partitions de multiplexer leurs minuteries avec les mêmes mécanismes que ceux employés dans le noyau, afin de fournir des services de gestion de l'horloge à un système d'exploitation de partition.

3.6.2 Gestion des interruptions et exceptions

XtratuM permet une gestion des interruptions et exceptions à fine granularité. Le gestionnaire d'interruptions et d'exceptions permet aux partitions de traiter des interruptions et exceptions durant leur période active.

Au niveau des interruptions, il est possible d'assigner chaque source à une partition particulière pour son traitement. L'assignation des sources du contrôleur d'interruptions du LEON est réalisée au niveau du PCS. Après le démarrage, il est impossible de réassigner les sources. Il est aussi possible d'assigner une source au noyau, afin que son assignation aux partitions soit proscrite. Lors du changement de contexte de partition, toutes les sources d'exceptions qui appartiennent à la partition sortante sont désactivées dans le contrôleur d'interruptions. En parallèle, les sources appartenant à la partition entrante sont activées. Un registre virtuel de statut des interruptions permet d'activer ou de désactiver le traitement des interruptions par une partition. Le traitement des interruptions est cependant toujours activé dans le noyau sauf dans les régions critiques où l'atomicité d'une opération doit être garantie.

En ce qui a trait aux exceptions, une table d'assignation des 256 sources d'exceptions disponibles sur les processeurs LEON est virtualisée afin de permettre la redirection des « traps » et autres exceptions, telles que les exceptions liées au module point flottant ou de protection de mémoire. Les exceptions matérielles sont traitées immédiatement par le moniteur de santé du système. Le cas échéant, une action de recouvrement est entreprise, soit en appelant un vecteur d'exception virtuel dans la partition ou en effectuant une action plus drastique, telle que le redémarrage de la partition.

XtratuM supporte aussi 32 interruptions « étendues » qui représentent des évènements virtuels provenant du noyau, tel que le déclenchement des minuteries et l'arrivée d'une nouvelle fenêtre d'activation de partition.

Lorsqu'une interruption survient, le noyau ne fait que mettre un drapeau à jour dans un registre virtuel de la partition en cours. Si le traitement des interruptions est activé dans la partition, un vecteur d'interruption virtuel est appelé. Cette virtualisation des interruptions évite de donner le contrôle direct du contrôleur d'interruptions aux partitions, afin de prévenir qu'une faute survenant dans une partition ne cause une défaillance complète du traitement des interruptions. Il est important de noter que le traitement des interruptions et des exceptions est nécessaire pour assurer le partitionnement robuste. Par exemple, sans interruption de minuterie pour forcer le noyau à prendre action, il en reviendrait aux partitions d'être assez « honnêtes » pour léguer le processeur au noyau lorsque nécessaire.

Le cheminement entre l'occurrence d'une exception ou d'une interruption et son traitement diffère quelque peu en fonction du type d'évènement. La logique globale de ce cheminement est bien résumée par l'exemple d'une interruption provenant d'un périphérique. La figure 3.7 présente cet exemple à l'aide d'un diagramme, en assumant que l'interruption est valide pour une partition et que les interruptions sont activées. Les étapes du traitement sont les suivantes :

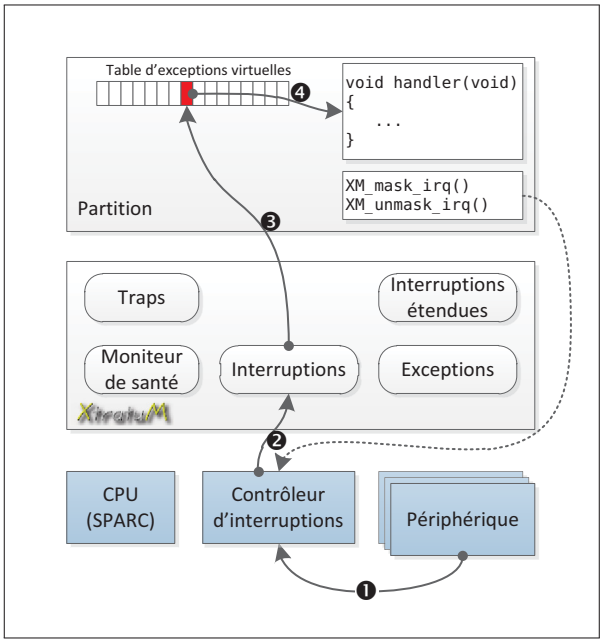


Figure 3.7 Cheminement de traitement d'une interruption de périphérique dans XtratuM

❶ Un périphérique donne son signal d'interruption au contrôleur d'interruption, signalant un évènement dont le traitement immédiat est nécessaire.

❷ Le contrôleur d'interruption interrompt le processeur, qui entre dans le gestionnaire d'interruptions du noyau XtratuM.

❸ Le gestionnaire d'interruptions détermine l'équivalence entre l'interruption matérielle et l'interruption virtuelle qui lui est associée dans le PCS. Le drapeau d'interruption virtuelle associé est activé. Un processus d'émulation d'interruption est démarré si l'interruption est valide pour la partition en cours selon le PCS. Sinon, le drapeau d'interruption virtuelle demeure activé et servira à démarrer un gestionnaire d'interruption différé lors de la prochaine activation de la partition cible.

❹ La partition en cours traite l'interruption virtuelle dans un gestionnaire d'interruptions virtuel dont l'adresse était spécifiée dans la table d'exceptions virtuelles.

Dans les autres cas d'exceptions et d'interruptions, les différences de traitement se situent principalement au niveau des structures internes du noyau qui sont consultées pour déterminer l'action à prendre.

3.6.3 Virtualisation des opérations en mode superviseur

L'architecture SPARC employée dans les processeurs LEON est dotée d'un mode de fonctionnement superviseur compatible avec les objectifs d'un système d'exploitation classique à gestion de mémoire virtuelle. Lorsque le processeur est en mode superviseur, toutes les opérations définies par l'architecture sont disponibles, incluant les opérations dites « privilégiées ». À l'inverse, lorsque le processeur est en mode usager, seules les opérations n'ayant aucun effet sur la protection des ressources sont permises. À titre d'exemple, l'accès direct aux registres de contrôle du processeur n'est pas disponible en mode usager, car les instructions pour l'accès à ces registres ne sont disponibles qu'en mode superviseur [1].

Puisque les partitions sont exécutées en mode usager, il leur est impossible d'affecter les registres de contrôle de la machine. Il est cependant nécessaire d'émuler certaines des caractéristiques du mode superviseur afin de permettre une programmation flexible des systèmes d'exploitation de partition. Pour pallier cette restriction, les caractéristiques superviseur du processeur sont virtualisées, c'est-à-dire qu'elles sont émulées en logiciel. Une série d'hyperappels permettent donc aux partitions de réaliser certains comportements de type superviseur de manière indirecte, en passant par un appel système au noyau. Il devient alors possible d'assurer la protection du système, car le noyau peut ainsi valider les requêtes faites par les partitions.

Le tableau 3.1 résume les opérations en mode superviseur qui sont virtualisées dans XtratuM et la méthode de virtualisation employée.

1. Les instructions `RDPSR` et `WRPSR` de l'architecture SPARCv8 sont dédiées à l'accès au registre de l'état du processeur (PSR, « Processor Status Register »).

Tableau 3.1 Méthodes de virtualisation des fonctions privilégiées dans le noyau XtratuM

Fonction privilégiée	Méthode de virtualisation
Gestion de l'horloge et des minuteries	Opérations de gestion de l'horloge et des minuteries indirectement à travers le noyau.
Gestion des interruptions	Registres de drapeaux d'interruptions virtuels et support dans le noyau pour interruptions virtuelles.
Définition de la table des vecteurs d'exceptions	Table des vecteurs d'exceptions virtuelle et registre d'adresse de base de la table (TBR, « Trap table Base Register ») virtualisé.
Entrées/sorties sur périphériques	Opérations de lecture et d'écriture des ports d'entrée/sortie indirectement à travers le noyau.
Gestion des fenêtres de registres SPARC	Gestion automatique du débordement par le noyau et opération de vidange des fenêtres indirectement à travers le noyau.

3.7 Mécanisme d'hyperappels

Comme nous l'avons mentionné précédemment, tous les services du noyau sont appelés à travers une interface centralisée qui passe à travers la barrière d'isolation vers le noyau. Le mécanisme de réalisation de cette interface est l'hyperappel.

Les hyperappels permettent d'accéder à tous les services du noyau à l'aide d'une syntaxe normale d'appels de fonctions. Cependant, au lieu de simplement effectuer un branchement vers une routine de librairie, les hyperappels emploient des déroutements (exceptions synchrones ou « traps ») afin d'entrer en mode superviseur. La convention d'appel à l'entrée d'une routine d'hyperappel est légèrement modifiée afin de transmettre un pointeur vers une structure d'information sur le contexte du processeur au moment du trap. Lorsque le processeur est en mode superviseur, un gestionnaire centralisé détermine la nature de l'hyperappel et redirige l'exécution vers une routine appropriée à l'aide d'une table de sauts (« jump table »).

Lorsqu'un hyperappel est effectué, le contexte du processeur est enregistré. Aux yeux du processeur, un trap d'hyperappel est traité de la même manière que les exceptions et les interrup-

tions normales. Une instruction de retour d'exception [2] est ensuite utilisée pour revenir de la routine d'hyperappel.

Les détails d'implémentation des hyperappels sont transparents au niveau du code source des partitions. Ils apparaissent comme des appels de fonctions normaux dans le code C. Le temps système perdu pour l'entrée et la sortie de l'hyperappel est cependant très élevé puisque le contexte du processeur est sauvegardé et que plusieurs détails liés à la protection du système en mode superviseur doivent être traités. La figure 3.8 présente le diagramme de séquence de l'hyperappel rapide `XM_get_time()` qui sert à obtenir une valeur d'horloge réelle ou virtuelle. Après l'entrée dans le gestionnaire d'hyperappel, la routine `GetTimeSys()` est appelée à l'intérieur du noyau pour traiter la demande de service.

Lors de la compilation, l'hyperappel `XM_get_time` est compilé en trap identifié numériquement. La figure 3.9 présente le résultat de la compilation du code de l'hyperappel tel qu'implémenté dans la librairie `libxm.a`. On y remarque que l'implémentation du service est opaque : seuls le trap et sa préparation sont implémentés dans `libxm.a`. L'utilisation d'un identifiant numérique défini par convention dans la documentation permet de découpler le noyau de la partition appelante, pour éviter que le code source du noyau ne soit considéré « lié » au code source de la partition. Ce découplage peut sembler inutile, mais il est nécessaire pour éviter de contaminer le code des partitions avec la licence GPL utilisée dans le noyau. Les développeurs originaux de XtratuM ont d'ailleurs clarifié dans un addenda de la licence qu'ils ne considéraient pas le déroutement vers les hyperappels comme étant un lien logiciel au sens de la licence GPL.

Puisque les hyperappels sont des exceptions synchrones, leur traitement s'effectue avec les interruptions désactivées. Ils sont donc considérés comme des régions critiques du point de vue du noyau. Cette caractéristique impose plusieurs ramifications fonctionnelles. Premièrement,

2. Dans l'architecture SPARCv8, l'instruction `RETT` combinée avec l'instruction `JMPL` dans l'emplacement de délai est employée pour revenir au programme interrompu après une exception, interruption ou trap.

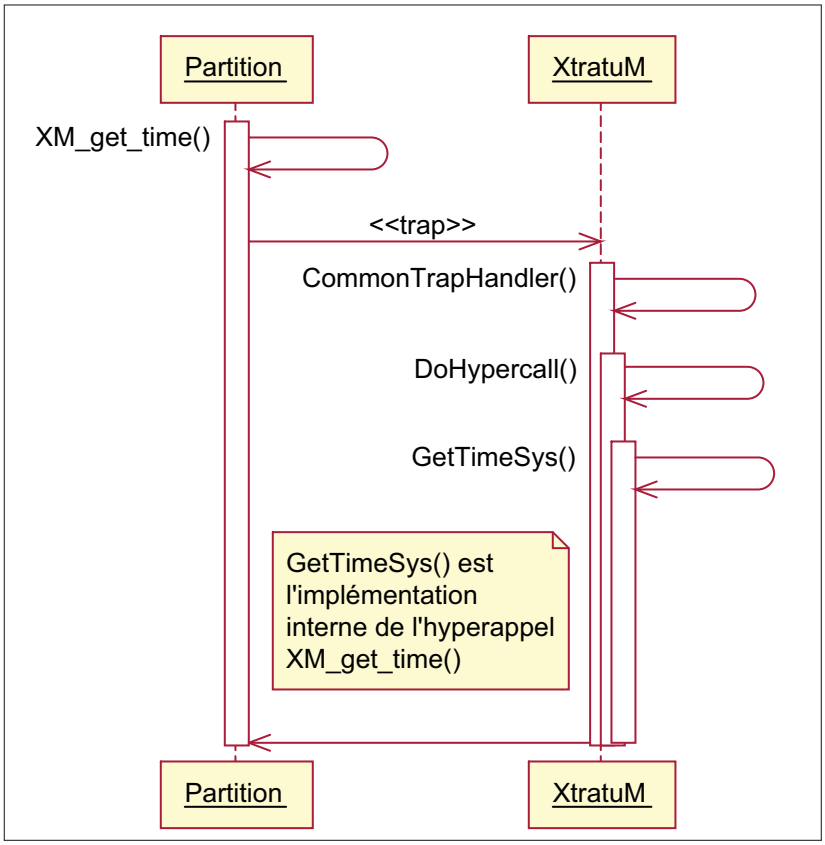


Figure 3.8 Diagramme de séquence pour l'hyperappel `XM_get_time()`

les hyperappels sont non-préemptibles, sauf par d'autres exceptions. Il est donc impossible de traiter des interruptions durant la durée d'un hyperappel, incluant les interruptions critiques, telles celles liées aux minuteries. La gigue temporelle liée aux interruptions est donc directement dépendante de l'hyperappel ayant le plus long délai d'exécution. Deuxièmement, en raison de cette non-préemptibilité, il est techniquement possible de causer un dépassement temporel d'une fenêtre d'activation de partition en effectuant un hyperappel très « coûteux » à l'instruction immédiatement précédant le moment où le changement de contexte aurait été déclenché par la minuterie système. L'utilisation du mécanisme d'hyperappels par le noyau XtratuM doit donc être compensée d'une manière ou d'une autre afin d'éviter les dépassements temporels qui causeraient la défaillance de l'isolation temporelle. Dans le manuel de XtratuM, il est indiqué que l'ordonnanceur prend en considération la durée de l'hyperappel

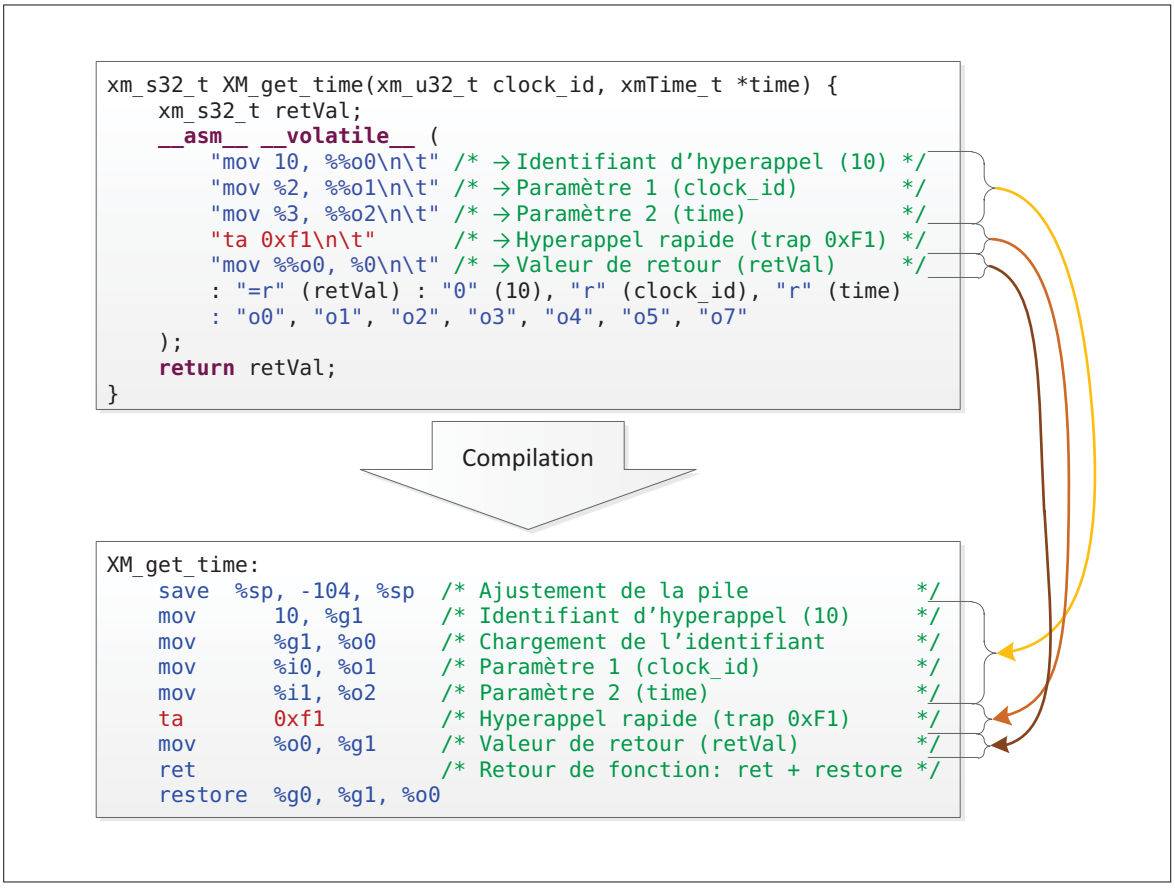


Figure 3.9 Code compilé de la routine d'hyperappel `XM_get_time()`

le plus long (son « worst case execution time » ou WCET) afin de le soustraire de manière transparente à la durée de chaque fenêtre d'activation. Il s'avère que ce mécanisme n'est pas implémenté dans le noyau. Le moyen alternatif afin de compenser l'effet de dépassement est de manuellement soustraire le WCET de l'hyperappel le plus coûteux à la durée nominale de chaque fenêtre d'activation du plan d'exécution dans le PCS. Cette compensation est illustrée à la figure 3.10, avec la contribution temporelle du temps d'exécution du changement de contexte entrant et sortant des partitions. Dans cette figure, on remarque que la fenêtre d'activation de la partition A doit être raccourcie de 1.5 ms pour compenser la durée des changements de contexte entrant et sortant, ainsi que la durée de l'hyperappel ayant le plus long délai d'exécution dans la partition. En réalité, ces délais sont beaucoup inférieurs à ce qui est

présenté dans la figure. Il n'en demeure pas moins que ces délais doivent être compensés par l'ajout de temps mort à la fin de chaque fenêtre d'activation. Le plan d'exécution de notre étude de cas de la section 5.10 présente d'ailleurs un exemple de compensation de la durée des fenêtres d'activation.

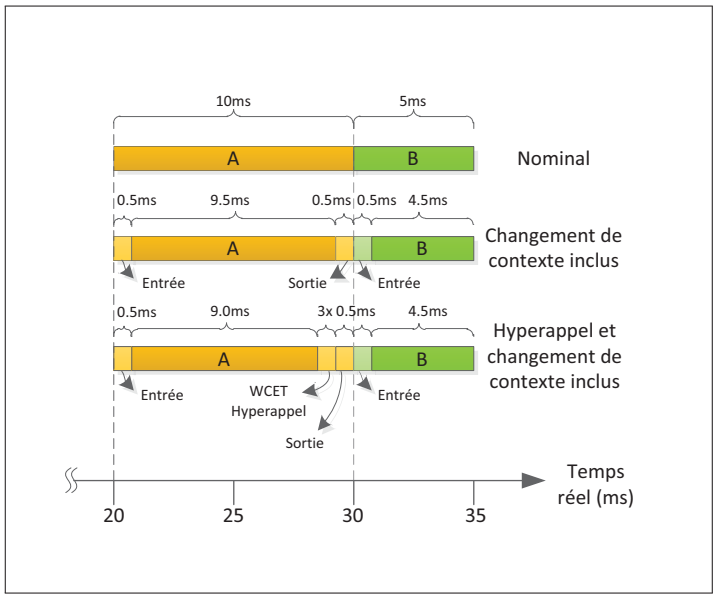


Figure 3.10 Compensation du WCET de l'hyperappel le plus coûteux dans le plan d'exécution

Finalement, il est important de mentionner que plusieurs des hyperappels de la librairie `libxm.a` sont en fait des pseudohyperappels. Ces pseudohyperappels sont des fonctions qui séparent un traitement complexe en une série d'hyperappels plus simples. Cette approche permet de simplifier le noyau en réduisant le nombre de fonctions individuelles à implémenter et valider. L'extrait 3.1 présente le code source du pseudohyperappel `XM_create_queuing_port()` utilisé pour ouvrir un port à file d'attente. Dans cet exemple, on remarque que les hyperappels `XM_open_object()`, `XM_ctl_object()` et `XM_close_object()` sont utilisés. Le pseudohyperappel `XM_create_queuing_port()` démontre aussi l'utilisation d'un mécanisme générique d'accès à des flux de données, basé sur des descripteurs d'objets. Ce mécanisme est utilisé dans XtratuM pour la réalisation des appels

d'IPC, du moniteur de santé du système et des pilotes matériels en utilisant seulement cinq hyperappels dans le noyau. Le modèle des flux de données est basé sur le modèle classique UNIX et les cinq fonctions de base décrites au tableau 3.2 sont utilisées pour accéder à ces flux.

```
xm_s32_t XM_create_queuing_port(char *portName, xm_u32_t maxNoMsgs,
        xm_u32_t maxMsgSize, xm_u32_t direction)
{
    xm_s32_t oDesc;
    struct xm_ctl_qport_cfg cfg;
    char name[256] = "~/comm/";

    strcat(name, portName);
    oDesc = XM_open_object(name, direction);

    if (oDesc < 0) return XM_INVALID_CONFIG;

    if (XM_ctl_object(oDesc, XM_COMM_GET_PORT_CFG, &cfg) != XM_OK)
    {
        XM_close_object(oDesc);
        return XM_INVALID_CONFIG;
    }

    if (cfg.maxMsgSize || cfg.maxNoMsgs)
    {
        if ((cfg.maxMsgSize != maxMsgSize) || (cfg.maxNoMsgs != maxNoMsgs))
        {
            XM_close_object(oDesc);
            return XM_INVALID_CONFIG;
        }
    }

    return oDesc;
}
```

Extrait 3.1 Code source de l'hyperappel `XM_create_queuing_port()`

Tableau 3.2 Fonctions de base du mécanisme de pilote générique de XtratuM

Nom XtratuM	Nom UNIX	Fonction réalisée
`XM_open_object`	`open`	Ouverture d'un flux de données
`XM_close_object`	`close`	Fermeture d'un flux de données
`XM_ctl_object`	`ioctl`	Modification des paramètres d'un flux de données
`XM_read_object`	`read`	Lecture à partir d'un flux de données
`XM_write_object`	`write`	Écriture vers un flux de données

3.8 Partitionnement temporel

Le partitionnement temporel de XtratuM est réalisé par un ordonnanceur cyclique de partitions basé sur le modèle décrit aux sections 2.3.1 et 2.3.1.2 de la norme ARINC-653. Le plan d'exécution est défini dans le PCS avec les paramètres de décalage et de durée pour chaque fenêtre d'activation. Comme dans le cas du partitionnement spatial, l'outil `xmcparser` s'occupe de réaliser une première validation des règles de partitionnement temporel. Ainsi, les erreurs telles que le chevauchement temporel ou l'absence d'une partition dans le plan d'activation sont attrapées avant la compilation du code. L'extrait 3.2 présente l'exemple du plan d'activation du tableau 2.1 tel qu'il apparaîtrait dans le PCS en format XML. L'attribut « partitionId » est associé à la définition des partitions, qui a été omise de cet extrait par souci de concision. Les valeurs de « partitionId » sont de « 0 » et « 1 » pour les partitions « A » et « B », respectivement.

```
<!−− Éléments précédents omis −−>
<Processor id="0" frequency="80Mhz">
    <Sched>
        <CyclicPlan>
            <Plan name="defaultCPU0" majorFrame="15ms">
                <!−− Chaque fenêtre d'activation (slot) comporte
                    un identifiant unique ("id"), un identifiant de
                    partition associé ("partitionId"), un décalage par
                    rapport au début du bloc d'activation majeur
                    ("start") et une durée d'activation ("duration") −−>
                <Slot id="0" partitionId="0" start="2ms" duration="3ms"/>
                <Slot id="1" partitionId="1" start="6ms" duration="6ms"/>
                <Slot id="2" partitionId="0" start="12ms" duration="3ms"/>
            </Plan>
        </CyclicPlan>
    </Sched>
</Processor>
<!−− Éléments suivants omis −−>
```

Extrait 3.2 Exemple de définition du plan d'activation
dans le PCS d'un système XtratuM

XtratuM supporte une résolution d'une microseconde pour toutes les spécifications temporelles. La gigue temporelle liée à l'activation de partition peut cependant être beaucoup supérieure à cette résolution minimale en raison des délais associés au changement de contexte.

Même sur un processeur à haute performance comme les LEON, le délai entre l'activation de la minuterie d'ordonnancement et la fin du changement de contexte de partition peut être de l'ordre de plusieurs dizaines de microsecondes.

L'ordonnanceur est conçu afin de forcer l'application du plan d'exécution, peu importe les circonstances. Une des minuteries du noyau est affectée à l'activation de l'ordonnanceur. Lorsqu'une fenêtre d'activation doit démarrer, l'ordonnanceur est activé par la minuterie. Il force ensuite le changement de contexte inconditionnellement, pour autant qu'un hyperappel ne soit pas en cours de traitement. Cette restriction due au mécanisme d'hyperappel a déjà été mentionnée à la section 3.7. Si une partition se trouve dans l'état « Halt » ou « Suspend », l'ordonnanceur donne le contrôle à une boucle de perte de temps dans le noyau, afin d'absorber le temps restant de la partition. Aucune autre partition ne peut se faire allouer le temps d'une partition suspendue, peu importe les circonstances. Il est possible pour une partition en fonctionnement normal, dans l'état « Boot » ou « Normal », de demander au noyau de suspendre l'activité jusqu'à l'occurrence d'un évènement telle une interruption, à l'aide d'un service du noyau. L'ordonnanceur est appelé lorsque des interruptions surviennent afin de continuer l'exécution d'une partition ainsi « endormie », si la fenêtre d'activation actuelle lui appartient.

Il n'y a aucune relation de priorité entre les partitions : toutes les partitions sont considérées comme égales. Au sein des partitions, il est cependant possible d'implémenter un système d'exploitation de partition. Ce système d'exploitation forme un deuxième niveau d'ordonnancement, au-dessus de l'ordonnanceur cyclique du noyau. La politique d'ordonnancement d'un système d'exploitation de partition peut être arbitraire, car il s'agit de code utilisateur. Dans la majorité des cas, on verra soit un exécutif cyclique, soit un ordonnanceur à priorité fixe.

La norme ARINC-653 définit le deuxième niveau d'ordonnancement de l'APEX avec un système de tâches (« processes » en anglais) aux opérations et attributs clairement définis [12, sec 2.3.2, 3.3.2]. En particulier, ARINC-653 définit une interface d'ordonnanceur préemptif

à priorité fixe aux caractéristiques très proches des systèmes d'exploitation temps réel classiques. La norme mandate la possibilité d'employer des tâches périodiques et apériodiques, mais elle spécifie aussi que l'implémentation interne peut être arbitraire pour autant que ces deux modes soient disponibles. Enfin, ARINC-653 définit les services qui doivent être disponibles aux tâches ainsi que les différents états d'exécutions qui doivent être supportés par l'ordonnanceur. Nous ne couvrirons pas la liste exhaustive de ces services. Puisque XtratuM n'est pas un système conforme à la norme ARINC-653, il n'était pas nécessaire d'implémenter le deuxième niveau d'ordonnancement mentionné précédemment. À notre connaissance, tous les systèmes d'exploitation avioniques commerciaux supportent plusieurs types d'ordonnancement de deuxième niveau, dont ARINC-653. La conformité à ARINC-653 ne requiert que le passage de tests au niveau des interfaces disponibles et la validation de leur implémentation selon la norme. Il est donc possible pour les différentes compagnies de construire un support de l'APEX au-dessus de n'importe quel système d'exploitation temps réel. Par exemple, l'approche employée dans le projet AIR démontre comment passer d'un système d'exploitation temps réel existant à une implémentation conforme ARINC-653 en implémentant uniquement les services requis et en adaptant l'ordonnanceur du noyau.

En plus de supporter un ordonnanceur cyclique robuste, XtratuM permet aussi la mitigation de la gigue temporelle. Il est possible de configurer le noyau avec ou sans support pour la cache. Les caches sont de loin les éléments causant le plus de difficultés pour l'analyse du WCET des composantes logicielles [21]. Il est cependant souvent nécessaire d'utiliser les caches afin d'atteindre les vitesses de traitement nominales des processeurs modernes, dont le cycle d'exécution des instructions est plusieurs fois plus rapide que le délai d'accès des DRAMs. Les technologies actuelles de DRAM, telles que DDR1, DDR2 et DDR3 posent un problème additionnel à ce niveau parce qu'elles présentent un compromis de bande passante élevée au détriment d'une latence élevée. XtratuM permet aussi de vidanger les caches lors du changement de contexte, ce qui réduit l'incertitude de délai d'exécution causé par la cache

au cas le plus pessimiste d'une cache vide. Cela réduit les performances moyennes, mais permet d'éviter de valider le logiciel dans un environnement où les défauts de caches sont trop optimistes ou lorsqu'ils dépendent des opérations effectuées dans le contexte de la partition précédemment active. La vidange de cache assure un état de cache identique—vide—à chaque fenêtre d'activation.

Finalement, une caractéristique présente dans les noyaux de partitionnement robustes ARINC-653 est la détection du dépassement d'échéances des tâches. Chaque tâche peut se voir associer un attribut de « budget temporel » qui agit comme une minuterie chien de garde (« watchdog timer » en anglais). Cette minuterie décompte le temps à partir d'un budget initial et elle est réinitialisée à l'aide d'un service APEX. Si la minuterie atteint un décompte de zéro, cela signifie qu'une échéance temporelle critique est dépassée et que le moniteur de santé du système doit être avisé. Le noyau XtratuM supporte des mécanismes similaires. En particulier, XtratuM fournit une minuterie chien de garde par partition, qui génère un évènement de moniteur de santé du système lorsqu'elle arrive à échéance. De plus, un système d'exploitation de partition sous XtratuM pourrait employer des minuteries virtuelles de partition pour réaliser la détection du dépassement des échéances à la granularité d'une tâche, comme dans ARINC-653.

3.9 Partitionnement spatial

Dans XtratuM, le partitionnement spatial est réalisé avec une combinaison de conventions, de support logiciel et de support matériel. XtratuM a été conçu pour utiliser les fonctionnalités de protection disponibles initialement dans le processeur Atmel AT697E, dans lequel le MMU de l'architecture SPARCv8 n'est pas implémenté. Pour cette raison, le modèle de partitionnement spatial est simple, mais aussi restreint.

Le noyau et les partitions fonctionnent tous en mode d'adressage réel puisqu'il n'y a pas de MMU pour permettre une multitude d'espaces d'adressage virtuel. En raison du fonctionnement en adressage réel, tous les binaires doivent être compilés et liés avec des adresses

statiques qui ne se chevauchent pas. Les zones mémoires du noyau et des partitions sont donc définies dans le plan de configuration du système (PCS) seulement avec leurs plages d'adresses physiques réelles.

À travers le système, l'isolation spatiale est réalisée à plusieurs niveaux :

- les partitions ne peuvent accéder qu'à leur propre espace mémoire ;
- les services du noyau doivent être demandés par hyperappel et il est impossible d'effectuer des appels directs de fonction dans l'espace mémoire du noyau ;
- les transactions d'entrée/sorties doivent passer par le noyau qui gère l'accès à l'espace d'adressage des entrées/sorties ;
- les transactions d'IPC doivent aussi passer par le noyau et les données sont copiées d'un espace mémoire de partition à un autre par le noyau ;
- les pointeurs passés aux hyperappels doivent tous être vérifiés afin de prévenir des accès interdits à la mémoire, réalisés en mode privilégié par empoisonnement de pointeur ;
- les partitions possèdent chacune leur propre pile ;
- le noyau possède une pile dédiée.

L'implémentation du partitionnement spatial débute à la validation du PCS par l'outil `xmc-parser`. C'est durant cette étape que les conventions liées au modèle de partitionnement spatial sont vérifiées. Les tailles et adresses des plages de mémoire sont validées en fonction de la configuration de la plateforme cible et des règles d'isolation. Les structures de données utilisées par le noyau dans son application du partitionnement spatial sont aussi construites par `xmcparser` et enregistrées dans la table de configuration binaire.

Au niveau du noyau, le partitionnement spatial est appliqué à l'aide de plusieurs mécanismes. La première ligne de protection est la protection matérielle de la mémoire. Ensuite, la validation des paramètres d'hyperappels permet d'attraper plusieurs fautes d'isolation spatiale qui

pourraient demeurer. Enfin, l'architecture du noyau elle-même est conçue afin de réduire les risques de fautes d'isolation.

La protection matérielle de la mémoire est réalisée avec des mécanismes matériels accessoires, plutôt que directement à l'aide d'un MMU, puisque ce dernier est absent dans certaines versions des processeurs LEON. Le noyau XtratuM emploie trois mécanismes différents pour réduire au minimum les accès mémoire interdits. En premier lieu, le module d'interface aux mémoires externes du LEON est configuré en mode de protection d'écriture. Cette fonctionnalité rudimentaire de protection mémoire permet de générer une exception lorsqu'une écriture survient à l'extérieur de la plage mémoire de la partition active. Les registres associés à cette protection d'écriture sont mis à jour lors du changement de contexte de partition. Des restrictions d'alignement et de taille sévères sont liées à ce mécanisme : toutes les zones de mémoire de partitions doivent être d'une taille supérieure à 32 ko et leur taille doit être une puissance de deux. En second lieu, le module d'interface d'entrée/sortie est configuré afin d'empêcher tous les accès à la zone de l'espace mémoire liée aux entrées/sorties sur des périphériques externes. Il est donc impossible pour les partitions de directement lire ou écrire des registres dans les périphériques externes. Enfin, un des quatre points d'arrêt matériels (« watchpoints » en anglais) est utilisé afin de restreindre tous les accès à la moitié supérieure de l'espace mémoire des LEON, soit la plage 0x8000_0000-0xFFFF_FFFF, qui est utilisée pour les périphériques PCI et quelques registres systèmes qui ne doivent jamais être accédés directement par les partitions. La figure 3.11 résume la carte mémoire d'un système à deux partitions et les mécanismes de protection mémoire utilisés pour chaque zone de mémoire.

Tous les mécanismes de protection matérielle de la mémoire sont désactivés lorsque le processeur traite les exceptions en mode superviseur. Le noyau peut ainsi accéder à toute la carte mémoire sans restrictions.

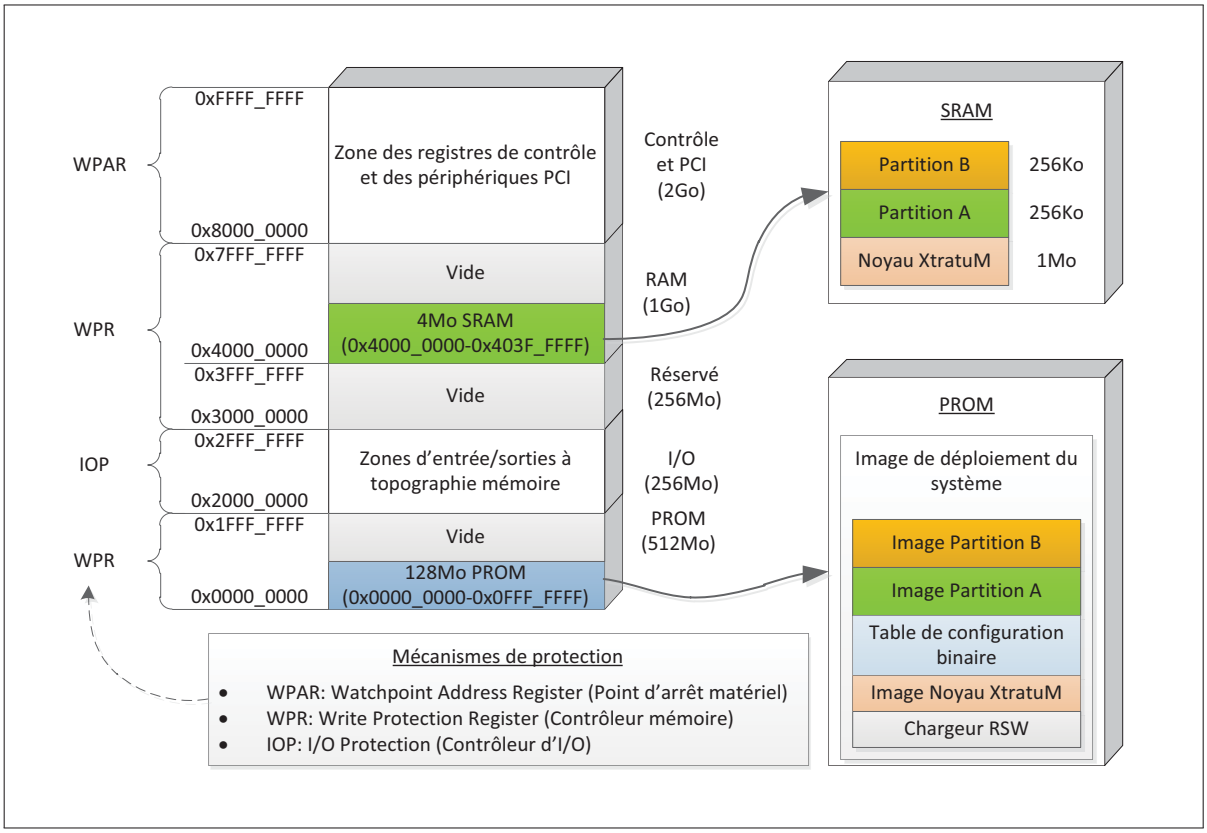


Figure 3.11 Exemple de carte mémoire d'un système XtratuM à deux partitions

En plus des mécanismes matériels, des mécanismes de validation logicielle permettent d'assurer l'isolation spatiale. Tous les hyperappels sont programmés de sorte à vérifier les paramètres qui leur sont passés. Les erreurs logiques et les valeurs erronées sont ainsi détectées. Les valeurs retournées par les hyperappels sont toutes signées. Une valeur négative souligne une erreur de traitement dont la signification peut être déterminée selon une liste prédéfinie.

Deux types de vérification additionnelle sont effectués au niveau des paramètres d'hyperappels. La première est au niveau des pointeurs, qui sont validés afin d'assurer qu'on ne demande pas d'opérations privilégiées par hyperappel dans des régions mémoires qui ne sont pas accessibles à une partition. On se souviendra que le noyau possède des droits d'accès à toute la carte mémoire sans restrictions. Sans cette vérification explicite des pointeurs, il serait possible, par exemple, de demander que le résultat d'une opération soit enregistré dans une partition autre

que la nôtre, ce qui empoisonnerait l'espace mémoire de la partition réceptrice. Le second type de vérification additionnelle est effectué dans les hyperappels d'accès aux entrées/sorties. XtratuM permet à plusieurs partitions de partager l'accès à un même registre si les bits accédés sont mutuellement exclusifs. Cela permet de simplifier les opérations telles que l'écriture et la lecture de ports d'entrée/sortie parallèles. Ainsi, certaines broches peuvent appartenir à une partition alors que d'autres broches affectées par le même registre appartiennent à une ou plusieurs autres partitions. Le noyau effectue automatiquement un masquage qui prévient toute modification intempestive des valeurs écrites dans les sections de registres appartenant à d'autres partitions. La configuration des ports d'entrées/sorties, à la granularité du bit, est effectuée dans le PCS.

Au niveau de l'architecture du noyau, plusieurs mécanismes sont en place pour assurer le partitionnement spatial robuste. Ainsi, une multitude d'assertions et de validations logiques (« sanity checks », en anglais) sont insérées à tous les niveaux dans le code source. Ces vérifications lors de l'exécution permettent de détecter des fautes dynamiquement. Certaines peuvent sembler inutiles, telles que des vérifications de valeurs de pointeurs sur des variables explicitement écrites par le noyau lors du démarrage. Ces vérifications sont cependant une assurance additionnelle du bon fonctionnement du noyau et forment une ligne de protection pour améliorer la fiabilité du partitionnement spatial.

Un autre aspect architectural du noyau qui améliore la fiabilité du partitionnement spatial est la vérification des zones mémoire du PCS lors du démarrage. Pour ce faire, le noyau emploie les mêmes contraintes que celles validées par `xmcparser` lors de la compilation du PCS en table de configuration binaire. Cette vérification, en conjonction avec d'autres validations du PCS, permet d'éviter qu'une table de configuration binaire corrompue ne permette le démarrage d'un système où les contraintes d'isolation seraient enfreintes. Il serait aussi possible d'effectuer une vérification similaire à chaque début de bloc d'activation majeur pour assurer une meilleure probabilité de détection de fautes, mais cela n'est pas implémenté dans XtratuM.

Finalement, toute l'allocation de mémoire du noyau XtratuM est statique. Il n'y a aucun appel à des fonctions d'allocation dynamique de mémoire, telle que celles de la famille `malloc()`. Il est ainsi impossible pour le noyau de dynamiquement dépasser sa propre capacité de mémoire lors de l'exécution. En conséquence, la taille des tableaux doit être préconfigurée à une valeur suffisante pour toutes les ressources du noyau. Puisque la certification se fait au niveau du livrable tel que compilé, cette restriction peut causer des problèmes si on doit recompiler le noyau pour corriger un manque de ressources. Il est donc conseillé de s'assurer d'une certaine marge de manoeuvre au moment de la configuration, quitte à réserver plus de mémoire que ce qui sera ultimement nécessaire.

Malgré le fait que XtratuM ait été conçu pour le partitionnement robuste, la version originale ciblant les LEON sans MMU souffre de deux lacunes majeures qui rendent cette version inadéquate : 1) l'absence de protection en lecture entre les partitions et le noyau et 2) l'absence de protection en lecture en dehors des zones implémentées.

Dans le premier cas, le mécanisme de protection contre l'écriture utilisé pour la protection des données des partitions et du noyau n'empêche aucunement une partition de lire les données du noyau et des autres partitions. Le mécanisme de protection par watchpoint ne permet pas une granularité assez fine pour protéger les partitions, d'où son utilisation uniquement pour la région de contrôle en mémoire haute (voir figure 3.11). Cette lacune ne pourrait pas causer la propagation d'une faute vers une autre partition, comme cela pourrait se produire si la protection d'écriture était manquante. La conséquence n'en est cependant pas moins grave : une partition réalisée par une tierce partie pourrait soutirer de l'information propriétaire à une autre. Ce scénario est surtout problématique dans le cas où une unité matérielle de test scellée est fournie à une équipe de développement. Le manque de protection en lecture permettrait de télécharger de l'information propriétaire même si la machine est physiquement protégée, ce qui mettrait en péril la propriété intellectuelle des différents fournisseurs dans un contexte d'avionique modulaire intégrée où les différentes fonctions sont réalisées par des concurrents.

Dans le deuxième cas, la lacune se trouve au niveau des lectures vers des régions de mémoire qui ne sont pas câblées. Dans ce cas, le mécanisme de protection contre l'écriture est encore inefficace. La conséquence est cependant différente. En effet, si une lecture en dehors des zones implémentées se produit, une exception de type « data access error » se produit. Cette exception n'est pas liée à des registres additionnels qui permettent d'obtenir de l'information à propos de l'opération ayant causé l'exception. Le comportement habituel, lors d'une « data access error », est d'assumer un état irrécupérable, car d'autres erreurs de bus peuvent causer cette exception. Malgré qu'il puisse être possible de déterminer la source de l'erreur par désassemblage autour de l'adresse de retour ou par inspection du contexte de processeur, cela n'est pas recommandé. En effet, si l'erreur était effectivement irrécupérable, l'inspection autour de l'adresse de retour pourrait causer une exception imbriquée, ce qui causerait un redémarrage dur de la machine. Si un MMU avait été utilisé, l'erreur d'accès aurait été détectable à ce niveau. Les conventions d'implémentation de MMU présupposent la présence de registres accessoires qui permettent dans ces cas de déterminer avec une grande précision la cause de l'exception, ce qui permet une récupération dans la plupart des cas.

Comme nous l'avons démontré, les conséquences de l'absence de MMU dans l'implémentation LEON de XtratuM sont majeures. Dans un système doté d'un MMU, il aurait été possible d'utiliser un espace mémoire « utilisateur » identique pour toutes les partitions, un espace mémoire « superviseur » assigné au noyau et un espace mémoire d'entrées/sorties, comme c'est le cas dans les systèmes d'exploitation traditionnels. Cela aurait grandement simplifié l'implémentation du partitionnement spatial robuste.

Une version avec support MMU de XtratuM était prévue pour les architectures LEON et IA-32 [3]. Nous n'avons cependant pas été en mesure d'en obtenir le code source, ni une confirmation de leur existence de la part des développeurs originaux. Nous verrons à la section 4.9.2

3. L'architecture IA-32 ou « Intel Architecture, 32-bit » est la famille d'architecture de processeurs dont l'origine provient du 80386 d'Intel. Cette architecture est actuellement beaucoup plus sophistiquée que les processeurs 80386 initiaux.

que notre implémentation multicoeur du partitionnement spatial emploie un MMU pour la réalisation de mécanismes de protection mémoire, ce qui permet d'éliminer les lacunes de la version originale de XtratuM.

3.10 Communication inter-partitions

Le modèle de communications inter-partitions de XtratuM est directement calqué sur le modèle ARINC-653 [46, p. 11]. Les partitions peuvent s'échanger des *messages* de taille variables, mais dont la taille maximale est prédéfinie. Ces messages proviennent d'un *port virtuel source* d'une partition pour se rendre à un ou plusieurs *ports virtuels destinations*, chacun associé à une partition différente.

Les messages sont transmis d'un port à un autre à travers des canaux virtuels en mode échantillonnage ou en mode à file d'attente. Un canal à échantillonnage, comme son nom l'indique, présente toujours le dernier message reçu, tel un échantillonneur-bloqueur. Un canal à file d'attente peut recevoir plusieurs messages jusqu'à l'occurrence de sa capacité maximale. Les messages sont extraits un par un lors de la lecture, dans l'ordre d'arrivée (« First In, First Out » en anglais, FIFO).

Dans XtratuM, seules les communications directement entre les partitions sont supportées avec le mécanisme d'IPC. Pour accéder à d'autres types de ressources, l'abstraction des pilotes génériques, présentée à la section 3.7, doit être utilisée.

Une des restrictions du modèle de partitionnement spatial robuste d'ARINC-653 est que les partitions ne peuvent en aucun cas partager de régions mémoires communes en mode d'écriture [12, sec 2.3.1]. Il est cependant possible, en théorie, de permettre à une partition de lire une zone tampon d'une autre partition. Dans ce cas, il n'y a qu'une seule partition qui écrit dans la zone partagée, ce qui respecte les restrictions d'ARINC-653. Cette méthode est utilisée pour l'implémentation sans copie (« zero-copy » en anglais) des mécanismes d'IPC dans

certains noyaux de partitionnement robuste, dont VxWorks 653. La méthode zero-copy est plus performante, car elle évite le cycle de copie qui passe à travers le noyau avec les méthodes habituelles. Elle est cependant aussi plus complexe à implémenter, car un protocole de cohérence doit être développé afin d'éviter qu'un changement de contexte de partition n'affecte la validité de la transaction. XtratuM ne supporte pas l'approche « zero-copy ». Toutes les transactions d'IPC s'effectuent avec copie dans XtratuM. De plus, ces transactions sont ininterruptibles, donc atomiques, car elles sont réalisées par hyperappel.

Les transactions d'IPC sont toujours en mode non bloquant dans XtratuM. Par exemple, lors d'une transaction de lecture de port, l'hyperappel retourne immédiatement même si aucune donnée n'est disponible. Cette caractéristique est l'inverse du comportement des ports de communication dans ARINC-653. La raison de cette différence est que les opérations d'IPC dans ARINC-653 sont liées à la *tâche* initiatrice. Les tâches font partie du second niveau d'ordonnancement d'ARINC-653 (voir section 3.8). Il est logique dans ce contexte de bloquer sur les opérations d'IPC, car un ordonnanceur préemptif à priorité fixe se servirait de cette opportunité pour débloquer une tâche d'un niveau de priorité inférieur. Dans le cas de XtratuM, les opérations d'IPC sont en mode non bloquant afin de permettre à l'implémentation de l'ordonnancement de second niveau de l'utilisateur de s'en servir sans risquer d'affecter la disponibilité de toute la partition. En contrepartie aux ports non bloquant, une interruption virtuelle est fournie à chaque partition afin de l'aviser de l'arrivée d'une donnée dans un de ses ports. La sémantique en mode bloquant au niveau des tâches comme dans ARINC-653 peut ainsi être implémentée par-dessus les mécanismes d'IPC en mode non bloquant de XtratuM. Les interruptions virtuelles de ports permettent d'invoquer l'ordonnanceur de deuxième niveau afin qu'il valide si une tâche prioritaire était bloquée sur un port.

3.11 Moniteur de santé du système

Le moniteur de santé du système (« Health Monitor » ou HM, en anglais) est un service critique dans les noyaux de partitionnement robuste. Sans ce service, les seules alternatives en cas de faute seraient le redémarrage des partitions affectées ou leur arrêt inconditionnel, faute d'une logique de gestion plus précise.

Le HM de XtratuM permet d'enregistrer un journal des fautes de chaque partition et de déterminer l'action à prendre pour chaque type de faute. Il est ainsi possible pour une partition de type superviseur d'observer la progression des fautes dans un système et de réagir spécifiquement à chaque instance en fonction d'une logique définie par le développeur du système. La complexité de la gestion des fautes est donc exportée à l'extérieur du noyau, simplifiant ainsi sa réalisation au minimum des fonctions nécessaire au traitement immédiat des fautes.

Dans XtratuM, le comportement par défaut de chaque exception du processeur est d'activer un évènement du HM. Il est aussi possible pour chaque partition et pour le noyau de manuellement générer des évènements. Lorsque le HM reçoit un évènement, il l'enregistre dans un journal et exécute ensuite une action préconfigurée en fonction d'une table de gestion d'évènements fournie dans le PCS. Le tableau 3.3 présente la liste des actions possibles.

Le cheminement entre l'occurrence d'une faute et son traitement par le HM est résumé par le diagramme de la figure 3.12. Les étapes du cheminement sont les suivantes :

❶ Un évènement du HM est créé lors de la détection d'une erreur ou d'une défaillance dans le noyau ou dans une partition. Cet évènement est transmis au noyau HM par le service interne `HmRaiseEvent()` ou par l'hyperappel `XM_trace_event()` associé à un évènement irrécupérable (`XM_TRACE_UNRECOVERABLE`).

❷ Le module de moniteur de santé du système dans XtratuM se sert d'une table de configuration des actions dans le PCS pour déterminer l'action à prendre.

Tableau 3.3 Liste des actions du moniteur de santé du système de XtratuM, adaptée de [46, sec 2.7.2] avec la permission de la licence GFDL

Action (`XM_HM_AC_`)	Description
IGNORE	L'évènement est ignoré.
PARTITION_SHUTDOWN	Une interruption virtuelle d'arrêt de partition est envoyée à la partition.
PARTITION_COLD_RESET	La partition est redémarrée à froid.
PARTITION_WARM_RESET	La partition est redémarrée à chaud.
PARTITION_SUSPEND	La partition est suspendue, c'est-à-dire qu'elle est mise à l'état « Suspend » en attente d'une réactivation de l'état normal par une partition de type superviseur.
PARTITION_HALT	La partition est stoppée, c'est-à-dire qu'elle est mise à l'état « Halt ». Elle doit ensuite être redémarrée par une partition de type superviseur.
SYSTEM_COLD_RESET	Le système au complet est redémarré à froid.
SYSTEM_WARM_RESET	Le système au complet est redémarré à chaud.
SYSTEM_HALT	Le système au complet est arrêté.
PROPAGATE	Aucune action n'est réalisée par le noyau : l'évènement est propagé vers la partition avec une exception virtuelle.

❸ L'évènement est enregistré dans le journal du HM s'il existe. Rien ne se produit si aucun pilote n'avait été associé au noyau dans le PCS pour l'enregistrement du journal. Le noyau XtratuM fournit un pilote de blocs de mémoire en file circulaire qui permet d'enregistrer le journal sans utiliser de périphériques externes.

❹ Le noyau effectue l'action au niveau de la structure de donnée de partition dans le cas des évènements de partitions ou globalement au niveau du système dans le cas d'un évènement du noyau.

❺ Si l'action était `XM_HM_AC_PARTITION_SHUTDOWN` ou `XM_HM_AC_PROPAGATE`, une exception virtuelle est générée et son gestionnaire est exécuté au niveau de la partition aussitôt que possible.

La table de configuration du HM dans le PCS est validée par l'outil `xmcparser` lors de la compilation du PCS en table de configuration binaire. Des valeurs par défaut sont four-

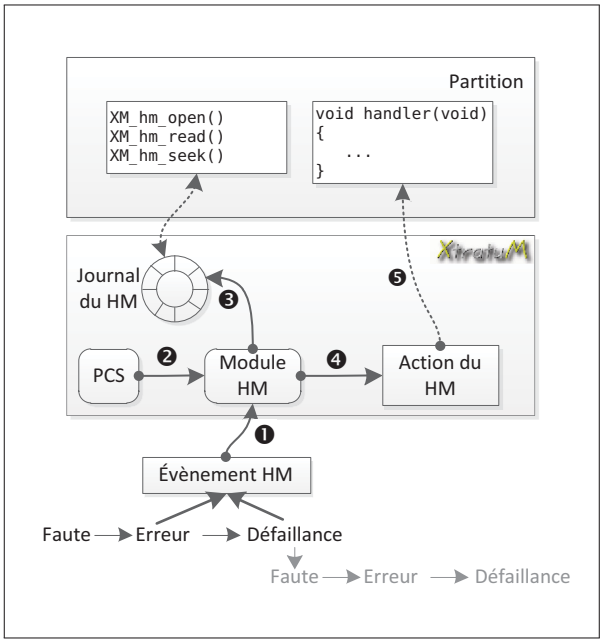


Figure 3.12 Cheminement d'une faute dans le moniteur de santé du système de XtratuM, adaptée de [46, p. 14] avec la permission de la licence GFDL

nies pour chaque évènement supporté par le noyau afin de réduire la taille du PCS. Il n'est nécessaire de fournir la configuration explicite d'un évènement que si l'on doit changer le comportement par défaut. La liste complète des évènements supportés est disponible à la table 2.2 du manuel de l'utilisateur de XtratuM [46, p. 15].

La version actuelle de XtratuM ne supporte actuellement pas les actions `XM_HM_AC_SYSTEM_COLD_RESET` et `XM_HM_AC_SYSTEM_WARM_RESET` au niveau du noyau. Ces deux actions résultent en l'arrêt du système, de la même manière que l'action `XM_HM_AC_HALT`, au lieu d'un redémarrage à froid ou à chaud, respectivement. Cela est dû au fait que le redémarrage est une opération propre à chaque plateforme matérielle. Il est cependant possible de modifier le code source du noyau pour réaliser le comportement de redémarrage désiré.

3.12 Récapitulation

Dans ce chapitre, nous avons présenté l'implémentation de tous les blocs fonctionnels du noyau XtratuM. Nous avons aussi couvert les outils de configuration du système et le déploiement du système partitionné.

XtratuM est un noyau de partitionnement robuste réalisé à l'aide du concept de l'hyperviseur. Ainsi, chaque partition est traitée comme une machine virtuelle indépendante et isolée des autres partitions par des mécanismes simples. Les partitions évoluent en mode non privilégié du processeur et doivent effectuer les opérations privilégiées à l'aide d'un mécanisme d'hyperappels. Les hyperappels réalisent entre autres la gestion des partitions et la communication inter-partitions.

Un système basé sur XtratuM est déployé à l'aide d'une image mémoire unique qui contient le noyau, les images de partitions et une table de configuration binaire obtenue par compilation du plan de configuration du système (PCS). Le PCS est un fichier de configuration XML validé et compilé par l'outil `xmcparser`.

L'isolation spatiale est assurée par une combinaison de moyens logiciels et matériels. La composante logicielle de l'isolation spatiale est la validation des paramètres d'hyperappels. La composante matérielle est l'utilisation des mécanismes de protection de mémoire des processeurs LEON afin d'empêcher l'écrasement des données de partitions entre elles. Cependant, comme nous l'avons présenté, les mécanismes matériels utilisés sont insuffisants, car ils ne permettent pas une isolation complète des espaces mémoire des partitions, en raison de l'absence d'unité de gestion de mémoire (MMU) dans la cible LEON.

L'isolation temporelle des partitions, quant à elle, est assurée par un ordonnanceur cyclique statique qui force le changement de contexte de partitions selon le plan d'exécution configuré. Les interruptions et exceptions sont aussi gérées afin d'éviter la subtilisation de cycles d'une partition envers une autre.

Finalement, le noyau XtratuM est doté d'un moniteur de santé du système complet qui permet d'exécuter des actions de contingence lorsque des fautes surviennent. En plus d'exécuter des actions spécifiques pour chaque exception, il est aussi possible d'employer des partitions superviseur pour traiter les données du moniteur afin de remettre en fonction des partitions qui pourraient avoir subi des défaillances.

Dans le prochain chapitre, nous présenterons les adaptations architecturales nécessaires pour passer de la version monocoeur originale de XtratuM à une implémentation supportant les processeurs multicoeurs.

CHAPITRE 4

ANALYSE DE L'ADAPTATION MULTICOEUR DE XTRATUM

4.1 Survol

Ce chapitre couvre les aspects génériques de l'adaptation du noyau XtratuM afin qu'il soit possible de déployer des partitions sur tous les coeurs d'un processeur multicoeur. Les détails d'adaptation présentés dans ce chapitre sont applicables en général à l'adaptation multicoeur de XtratuM et ne dépendent pas de l'architecture PowerPC, ni des détails d'implémentation d'une réalisation particulière d'une architecture multicoeur.

Lors du passage d'une architecture de noyau adaptée aux processeurs monocoeurs vers une architecture supportant plusieurs coeurs identiques, nous avons rencontré une panoplie d'obstacles. Ces obstacles couvrent tout un spectre de catégories et plusieurs d'entre eux découlent directement de l'architecture du noyau XtratuM. Nous avons tenté de mettre tous ces problèmes en perspective par rapport aux aspects génériques du partitionnement robuste qu'ils affectent. L'identification de ces problèmes est un des trois objectifs principaux de nos travaux. Nous fournissons aussi des pistes de solution pour la résolution de la majorité des obstacles, car sans ces solutions, nous n'aurions pas été en mesure de réaliser le prototype du noyau adapté.

Nous traitons méthodiquement de l'adaptation de XtratuM en suivant une progression logique des composantes. Nous débutons avec les outils et les services de base pour ensuite couvrir toutes les composantes du noyau de sorte que chaque nouvelle composante adaptée repose sur l'adaptation des composantes précédentes.

Les spécificités techniques du port de XtratuM vers l'architecture PowerPC seront traitées au chapitre 5. L'adaptation architecturale de toutes composantes de XtratuM présentée dans

ce chapitre est préalable à tout l'effort de portage vers les PowerPC multicoeurs. Cependant, nous ferons référence à notre port PowerPC, nommé XtratuM-PPC, lorsque des éléments de l'adaptation seraient difficiles à saisir sans avoir une cible de référence.

4.2 Support existant pour traitement parallèle

Le code source du noyau XtratuM 2.2.2 est doté de plusieurs caractéristiques de support au partitionnement multiprocesseur. D'après des communications privées avec les développeurs originaux, une adaptation multicoeur vers les processeurs de l'architecture IA-32 était prévue. De même, un support pour le MMU de cette architecture était aussi planifié. Les auteurs originaux n'ont cependant pas encore réalisé une version multiprocesseur de leur noyau, ni publié de résultats à cet effet. Lors de nos dernières communications privées, nous n'avons pas été en mesure d'obtenir d'éclaircissements sur l'avancement de ces travaux. De plus, dans une révision de leur site web en date du 21 décembre 2010, les auteurs ont retiré la mention de cette version IA-32.

Une révision complète du code a permis de déterminer quelles adaptations existantes pouvaient être réutilisées. Au fil des sections d'adaptation, nous mentionnerons les éléments réutilisés du support multiprocesseur déjà présent dans le noyau XtratuM 2.2.2 pour LEON. Nous pouvons d'ores et déjà affirmer que ces éléments existants étaient insuffisants à la réalisation d'une version multicoeur complète et que plusieurs éléments importants sont manquants. Le support multiprocesseur existant se résume principalement à une duplication des structures de données au niveau de chaque processeur logique et l'ajout d'attributs d'identification du processeur à quelques structures de données du PCS. Notamment, l'implémentation des mécanismes de synchronisation multiprocesseur et de verrouillage de données partagées est entièrement absente.

4.3 Méthodologie d'adaptation

L'adaptation de l'architecture de XtratuM a été réalisée en suivant une méthodologie d'analyse itérative.

En premier lieu, nous avons déterminé les hypothèses et suppositions en fonction du problème de recherche et de nos objectifs tels que précisés aux sections 1.1 et 1.2, respectivement. Ces hypothèses et suppositions, présentées à la section 4.3.1, ont guidé toutes les phases de l'analyse et de l'adaptation.

Nous avons par la suite identifié les besoins d'une version multicoeur du noyau XtratuM. Nous présentons à la section 4.3.2 un résumé des besoins principaux. Les spécifications fonctionnelles et non fonctionnelles découlant de ces besoins ont fait l'objet d'une analyse exhaustive dans un document interne au projet AREXIMAS. L'adaptation du noyau présenté dans ce chapitre est une synthèse de ces travaux d'analyse et des tâches d'implémentation subséquentes.

La distinction entre le travail d'adaptation architecturale et le travail d'implémentation est importante. En effet, l'adaptation architecturale du noyau est indépendante du travail de portage. Après l'adaptation, il est possible de réaliser une version multicoeur de XtratuM sur n'importe quelle architecture de processeurs multicoeurs qui respecte les suppositions émises ci-après. La seule différence entre ces versions pour différentes architectures matérielles se trouvera au niveau de l'implémentation des routines de bas niveau qui réalisent l'abstraction de la couche matérielle. L'implémentation PowerPC de l'adaptation architecturale générique est présentée au chapitre 5. Lors de la présentation de l'adaptation, nous illustrerons certains aspects à l'aide de concepts provenant de l'architecture PowerPC. Ces illustrations n'empêchent toutefois pas un portage de l'architecture adaptée à d'autres familles de processeurs.

4.3.1 Hypothèses et suppositions

Notre adaptation de XtratuM est un prototype de noyau de partitionnement robuste multicoeur. Pour cette raison, nous émettons plusieurs hypothèses et suppositions qui restreignent la portée de l'effort d'adaptation.

Au niveau du processeur, l'adaptation dépend des suppositions suivantes :

- L'architecture cible implémente un modèle de parallélisme à multiples instructions et multiples données (MIMD selon la taxonomie classique de Flynn) avec mémoire partagée symétrique (« Symetrical Multiprocessing » ou SMP, en anglais) à accès uniforme [1] (« Uniform Memory Access » ou UMA, en anglais) dont la représentation schématique est fournie à la figure 4.1.
- L'architecture cible est dotée d'un MMU sur chaque coeur et ce dernier :
  - permet l'assignation de pages de taille fixe ;
  - permet l'assignation de blocs de tailles fixes dont la taille est supérieure à celle d'une page ;
  - est dotée d'un « translation lookaside buffer » (TLB [2]) ;
  - est dotée d'un mode de gestion des tables de pages par logiciel (« software table-searching ») ou d'un mode de chargement direct du TLB par logiciel.
- L'architecture cible supporte un protocole de cohérence de caches matériel.
- L'architecture cible est dotée d'un mécanisme de barrière globale permettant d'assurer l'ordonnancement des accès à la mémoire partagée entre les coeurs.
- L'architecture cible est dotée d'un compteur d'horloge locale à haute résolution indépendant sur chaque coeur.

1. Notons que le modèle MIMD SMP UMA est celui le plus largement répandu parmi les processeurs commerciaux au moment de la rédaction de ce mémoire.

2. Le TLB est la cache locale d'un MMU qui regroupe les descripteurs de pages les plus récemment utilisés pour rendre plus rapide la translation des adresses virtuelles vers les adresses réelles.

Ces suppositions sont compatibles avec plusieurs architectures multicoeurs présentement disponibles, dont l'architecture PowerPC 32-bit version 2.06 [54] telle que définie par Power.org et employée dans tous les processeurs PowerPC multicoeurs actuels. Les processeurs employant le modèle SMP-NUMA (NUMA ou « Non-uniform memory access »), comme la série Opteron d'AMD, sont potentiellement utilisables dans notre contexte, mais nous n'avons aucunement évalué les problèmes potentiels de l'accès non uniforme (NUMA) de la mémoire.

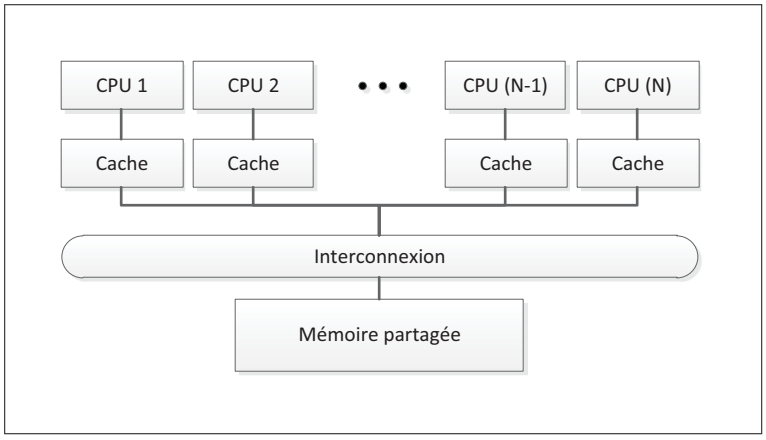


Figure 4.1 Digramme schématique d'une
organisation de système parallèle SMP UMA

Au niveau de la plateforme matérielle cible, l'adaptation repose sur les suppositions suivantes :

- La plateforme est dotée d'un gestionnaire matériel d'interruptions multicoeurs compatible avec la norme OpenPIC [2].
- Une quantité de mémoire RAM suffisante est disponible pour le chargement en mémoire physique de toutes les composantes logicielles, incluant le noyau, les partitions et leurs données, sans devoir recourir à un mécanisme de mémoire virtuelle à permutation de pages vers une mémoire de masse.
- Un micrologiciel d'initialisation matérielle de la plateforme est exécuté au démarrage afin de préparer la machine au chargement en mémoire du binaire du système.

Finalement, notre adaptation est sous-tendue par l'hypothèse que le modèle de partitionnement multicoeur développé n'a pas besoin d'être directement déployable dans un système à haut niveau de sûreté et qu'un prototype fonctionnel du modèle est suffisant.

4.3.2 Identification des besoins

Préalablement à l'adaptation du noyau XtratuM, nous avons identifié les besoins d'un noyau de partitionnement robuste multicoeur. Ces besoins sont énoncés sous forme d'extensions des fonctions de base du noyau XtratuM. Les besoins initiaux ayant mené à la conception actuelle du noyau XtratuM original ne sont pas répétés, puisqu'ils sont précurseurs à nos travaux.

Les besoins additionnels d'un noyau XtratuM multicoeur sont les suivants :

- Le module de partitionnement spatial du noyau devra supporter la protection de mémoire matérielle. Les caractéristiques suivantes devront être supportées :
  - protection indépendante des régions de code et de données ;
  - assignation de pages de données de taille native aux partitions, jusqu'à concurrence du nombre disponible dans le TLB après réservation par le noyau ;
  - assignation de blocs d'adresses de données plus grands que les pages de données, jusqu'à concurrence du nombre disponible dans le TLB après réservation par le noyau ;
  - détection du dépassement de pile des partitions.
- L'ordonnanceur de partitionnement temporel devra permettre le déploiement de plans d'exécutions avec des partitions exécutées sur chaque coeur disponible.
- L'ordonnanceur de partitionnement temporel devra supporter autant les plans d'exécutions SMP, AMP et hybrides.

- Tous les aspects de XtratuM liés à la gestion et à la validation du PCS, autant dans les outils que dans le noyau devront être adaptés afin de supporter les nouveaux besoins liés au partitionnement robuste multicoeur.
- L'outil de configuration des PCS devra supporter les spécificités, contraintes et pilotes des processeurs PowerPC en mode monocoeur et multicoeur.
- L'outil de configuration du code source du noyau devra supporter les paramètres des plateformes cibles supportées.
- Le noyau devra pouvoir démarrer un processeur avec un nombre arbitraire de coeurs, jusqu'à concurrence d'au moins 8 coeurs actifs, sans requérir d'intervention extérieure ou de configuration extérieure au PCS.

Le travail d'adaptation décrit dans ce chapitre réalise toutes les spécifications génériques découlant de ces besoins additionnels. Les spécifications en tant que telles sont paraphrasées lorsque nécessaires, mais ne sont pas autrement répétées ou justifiées.

4.4 Plan de configuration du système

Le plan de configuration du système est une structure de données globale du système, accédée en lecture seule. Lors de l'exécution du système, aucun élément de cette configuration ne peut être modifié. Les adaptations du PCS nécessaires au support d'un modèle de partitionnement multicoeur se retrouvent donc uniquement au niveau du contenu de la structure de donnée. Les éléments du PCS devant être adaptés relèvent de la réalisation de chaque module du noyau. Les éléments du PCS ainsi ajoutés ou modifiés seront couverts dans les sous-sections liées à chaque module du noyau.

L'adaptation de l'outil de validation et de compilation du PCS, `xmcparser`, se résume à l'ajout du support des nouveaux éléments et attributs de configuration, ainsi que des contraintes qui leur sont reliées. L'architecture de l'outil en tant que tel n'est aucunement affectée par l'ajout d'un support multicoeur.

L'outil `xmcparser` joue un rôle important dans l'adaptation multicoeur de XtratuM. En effet, plusieurs des additions fonctionnelles impliquent des contraintes de configuration du PCS qui doivent être validées par `xmcparser`. Lors de la validation de ces contraintes, certaines structures de données accessoires sont aussi générées, en plus de la table de configuration binaire. Ces structures de données accessoires sont utilisées lors de la construction des partitions. La majorité des ajouts de contraintes proviennent du nouveau modèle de partitionnement spatial parallèle. La section 4.9.2 présente le détail des ajouts au PCS et à `xmcparser` liés au partitionnement spatial parallèle.

4.5 Déploiement du système partitionné

Le déploiement d'un système XtratuM multicoeur est identique à la version originale monocoeur. Puisque les différents modes de partitionnement spatial et temporel parallèle emploient toujours l'abstraction de partitions individuelles, il n'est pas nécessaire d'avoir un chargeur de partitions parallèle. Le chargeur RSW demeure inchangé entre XtratuM et XtratuM-PPC, sauf bien sûr au niveau des routines de bas niveau de démarrage qui doivent être adaptées au processeur cible. La logique de chargement est cependant identique. La seule restriction additionnelle est que le chargeur doit absolument s'exécuter en mode monocoeur avec tous les autres coeurs désactivés, afin de prévenir l'exécution de code erroné sur les autres coeurs en attendant la fin du chargement. Cette restriction est implicite puisqu'il est évident que les autres processeurs ne doivent pas être démarrés avant que leur programme n'ait été chargé en mémoire par un processeur maître.

4.6 Services de base du noyau

Plusieurs adaptations importantes sont nécessaires au niveau des services de base de XtratuM afin de supporter le partitionnement parallèle. Cette sous-section décrit l'adaptation des services de base du noyau.

4.6.1 Duplication du fil d'exécution du noyau

Pour que le noyau XtratuM puisse supporter le partitionnement robuste avec une gestion centralisée des ressources, il est nécessaire de l'exécuter en mode de traitement parallèle symétrique (« Symetrical Multiprocessing » ou SMP en anglais) sur tous les coeurs. Par la suite, les différents types de plans d'exécution multicoeur (SMP, AMP, hybride) peuvent être gérés en synchronie. Notons que la terminologie « SMP » et « AMP » peut porter à confusion en raison de l'usage qui varie selon la source. Dans nos travaux, nous considérons que le noyau est exécuté de manière symétrique (SMP), car chaque coeur partage les mêmes régions de mémoire pour le code et les données. De plus, le démarrage, la configuration du matériel et la synchronisation des coeurs sont centralisés vers un seul coeur, donc il n'y a qu'un seul système d'exploitation logique qui contrôle tous les coeurs. Lorsque nous décrirons les plans d'exécution SMP, AMP et hybrides à la section 4.8.1, cela n'impliquera pas un changement de mode d'exécution du noyau de partitionnement robuste dans son ensemble. Il s'agira plutôt de modèles d'exécution qui pourront être supportés par le noyau exécuté en mode SMP.

L'exécution SMP du noyau est réalisée par la duplication du fil d'exécution principal du noyau sur chaque coeur. Des mécanismes de synchronisation comme les barrières et les verrous sont ensuite utilisés pour synchroniser les instances indépendantes sur chaque coeur. La duplication des fils d'exécutions (« forking ») du noyau s'effectue lors du démarrage.

Pour permettre une exécution SMP, le noyau doit avoir un contexte d'exécution complet sur chaque coeur. Ce contexte doit inclure les éléments suivants :

- une pile ;
- une structure de données de fil d'exécution ;
- une structure de données regroupant les horloges locales ;
- une structure de données locales du coeur avec un identifiant de coeur et les structures de données liées au MMU local ;

- une structure de données de configuration d'ordonnancement pour les partitions qui appartiennent au coeur courant.

Ces données de contexte sont appelées les « données locales » des coeurs, car chacun a besoin de sa propre copie pour gérer l'état des ressources qu'il contrôle.

Le noyau XtratuM monocoeur était déjà adapté à ce niveau pour une implémentation multi-coeur ultérieure. Les auteurs originaux avaient réalisé l'accès local par coeur en utilisant un mécanisme d'allocation de type « thread-local storage ». En thread-local storage, chaque accès à une structure de données locale est accompagné d'une requête d'indexage en fonction de l'identifiant du fil ou du coeur qui veut obtenir l'accès. La requête d'indexage est implicite ou explicite, dépendamment de l'implémentation. Dans le cas de XtratuM, l'indexage est réalisé explicitement à l'aide d'un identifiant de coeur.

4.6.2 Démarrage du noyau

Le démarrage du noyau en mode monocoeur suit les étapes suivantes :

1. La machine démarre.
2. Le chargeur RSW charge le noyau et les partitions en mémoire.
3. Le noyau XtratuM démarre et exécute ces étapes de démarrage avant le début de l'ordonnancement :
    a) initialisation des périphériques principaux du système ;
    b) initialisation du contrôleur d'interruptions ;
    c) initialisation des pilotes et objets du noyau ;
    d) initialisation des partitions ;
    e) démarrage des partitions.
4. L'ordonnanceur du noyau démarre et le système devient entièrement autonome.

Dans la version SMP du noyau, il est nécessaire d'ajouter des étapes au démarrage et de réorganiser la séquence afin de synchroniser correctement les évènements. La majeure partie de l'initialisation doit s'exécuter sur un seul coeur, de manière entièrement séquentielle, avant de relâcher les autres coeurs. Si tous les coeurs étaient relâchés dès le démarrage, chacun essaierait de réaliser les mêmes étapes, avec un ordonnancement légèrement différent. Il s'en suivrait une course de réinitialisation de chacun des registres de périphériques et le système se retrouverait dans un état indéterminé.

Les systèmes d'exploitation multicoeur traditionnels, tels que Linux ou FreeBSD, procèdent de la même manière : le coeur principal de la machine démarre le système d'exploitation et configure les périphériques avant de relâcher les autres coeurs du système. Par la suite, les autres coeurs exécutent chacun une version minimale du démarrage où l'initialisation des périphériques est omise et seulement leurs données locales sont initialisées. Enfin, les coeurs peuvent tous exécuter le même code pour autant qu'ils soient synchronisés lorsqu'ils accèdent à des ressources partagées.

Nous avons adapté le démarrage de XtratuM afin qu'il respecte un démarrage séquentiel sur le coeur principal, suivi d'un relâchement des autres coeurs avec un flot de démarrage réduit. Le pseudocode du démarrage adapté est présenté à l'algorithme 4.1. La fonction `Setup()` est le point d'entrée du noyau, immédiatement après le démarrage d'un coeur. On remarque dans cet algorithme que toute l'initialisation de bas niveau est réalisée par le coeur principal [3], dont l'identifiant est 0. Lorsque les autres coeurs démarrent et exécutent le même code, ils sautent par-dessus cette section et n'initialisent que les éléments locaux.

La synchronisation du démarrage est réalisée à l'aide de barrières globales, nommées `WAIT_BARRIER()` dans le pseudo-code. Ces barrières globales permettent d'assurer un ordre spécifique des opérations. Dans notre adaptation, les barrières n'ont pas de mécanisme

3. Dans un système multicoeur à N coeurs, ces derniers sont habituellement identifiés séquentiellement avec un identifiant dans l'intervalle $[0 \ldots N-1]$.

Algorithme 4.1 Démarrage SMP du noyau XtratuM

```
function Setup()
    IF get_cpu_id() == 0 THEN
       initialize_low_level_peripherals()
       initialize_IRQ_controller()
       initialize_kernel_objects()
       initialize_scheduler()
       /* Démarrage des autres coeurs */
       release_other_cores()
    END IF
    /* Attente du démarrage des autres coeurs */
    WAIT_BARRIER()
    /* Initialisation de l'horloge principale */
    IF get_cpu_id() == 0 THEN
       initialize_system_clock()
    END_IF
    WAIT_BARRIER()
    /* Initialisation des données locales */
    initialize_local_data()
    /* Initialisation des partitions */
    IF get_cpu_id() == 0 THEN
       setup_partitions()
    END IF
    WAIT_BARRIER()
    boot_partitions()
    WAIT_BARRIER()
    /* Synchronisation des horloges locales */
    synchronize_clocks()
    /* ----- GO ! L'ordonnancement démarre ici */
```

de délai d'attente maximal. Ainsi, un problème peut survenir si un des coeurs plante avant d'atteindre la barrière : les autres coeurs attendraient indéfiniment et le système complet planterait. Cependant, des adaptations logiques importantes seraient nécessaires afin d'ajouter la détection de cette condition. Une alternative est l'emploi d'une minuterie chien de garde qui ne serait jamais réinitialisée dans la boucle d'attente de la barrière. Il pourrait ainsi y avoir une logique de détection d'erreur à ce niveau, qui empêcherait le plantage dur de la machine.

Cependant, même si un délai d'attente maximal ou un chien de garde était implémenté, une défaillance lors du démarrage d'un coeur rendrait tout de même l'état du système invalide, même si les autres coeurs n'étaient pas en attente infinie, car le plan de configuration du système pourrait contenir un plan d'exécution nécessitant la présence de tous les coeurs. Des recherches additionnelles sont nécessaires afin de déterminer les adaptations permettant de supporter une défaillance de coeur au démarrage.

Le chronogramme de la figure 4.2 présente la séquence des évènements du démarrage SMP jusqu'au début de l'ordonnancement.

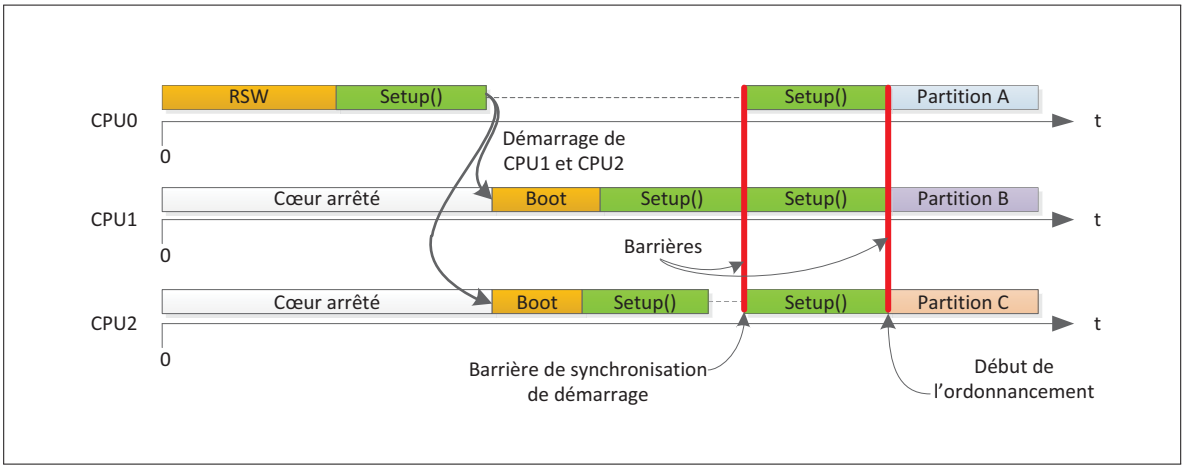


Figure 4.2 Chronogramme de la séquence de démarrage du noyau XtratuM sur trois coeurs

4.6.3 Gestion de l'horloge

L'horloge globale du noyau XtratuM sur LEON est utilisée autant par le noyau que par les partitions. Dans le cas de l'adaptation multicoeur, il serait possible d'utiliser une horloge globale si une telle ressource était disponible. Malheureusement, la plupart des processeurs multicoeurs ne sont pas dotés d'une horloge à haute résolution globalement accessible par tous les coeurs. Ce genre de périphérique peut cependant être réalisé à l'extérieur des coeurs par un compteur matériel lisible sur un bus partagé.

Afin de simplifier l'implémentation et d'augmenter la précision de la gestion de l'horloge, nous utilisons dans l'adaptation multicoeur une horloge locale et une minuterie matérielle locale sur chaque coeur. Une synchronisation entre les horloges locales au moment du démarrage du noyau est effectuée pour assurer un minimum de différences entre les temps perçus par les coeurs. Cette synchronisation est la dernière opération dans l'algorithme 4.1 et elle est nécessaire afin de permettre l'exploitation de groupes de partitions parallèles qui seront présentés plus loin à la section 4.8.1. Puisque tous les coeurs sont alimentés par la même horloge principale synchrone, leurs horloges et minuteries locales demeurent alignées après la synchronisation. Il peut rester une erreur résiduelle minimale entre les horloges locales des coeurs, mais cette dernière sera constante.

L'utilisation d'une horloge locale et d'une minuterie locale sur chaque coeur présente certains avantages :

- il n'y a aucune contention pour l'accès à la valeur de l'horloge ;
- la lecture de l'horloge locale dans un registre interne du processeur est immédiate alors que la lecture sur un périphérique peut prendre plusieurs cycles qui introduisent une incertitude additionnelle sur la valeur ;
- la corruption de la valeur d'horloge locale ou de minuterie locale d'un coeur n'a aucun effet sur l'ordonnancement des autres coeurs ;
- chaque coeur peut employer son propre ordonnanceur d'évènements locaux, ce qui élimine plusieurs couplages entre les coeurs.

Par contre, deux inconvénients découlent aussi de cette approche :

- la synchronisation globale d'évènements sans l'utilisation d'interruptions inter-coeurs est toujours accompagnée d'un délai de latence égal à la différence maximale de valeur d'horloge entre les coeurs qui essaient de se synchroniser, en raison de l'erreur systématique de synchronisation initiale des horloges locales ;

- la synchronisation des horloges peut prendre beaucoup de temps.

Le délai de synchronisation des horloges dépend de l'algorithme utilisé et il peut être élevé. Nous reviendrons sur ce problème en même temps que les détails d'implémentation de notre version sur PowerPC à la section 5.4.3.

4.6.4 Gestion des interruptions et exceptions

Les interruptions en mode multicoeur sont traitées de la même manière que celles en mode monocoeur : chaque coeur traite les interruptions qu'il reçoit. Il est important que toutes les sources d'interruptions matérielles, sauf celles conçues pour la messagerie ou la synchronisation inter-coeurs, ne soient assignées qu'à un seul coeur. Si plusieurs coeurs essaient en même temps d'accéder à un périphérique, il risque d'y avoir des conditions de course graves qui pourraient engendrer des fautes.

L'adaptation multicoeur de XtratuM gère le contrôleur d'interruption OpenPIC afin que le routage de chaque source soit destiné à un seul coeur. Les règles d'allocation des sources d'interruptions aux partitions sont validées à partir du PCS par l'outil `xmcparser`.

En plus de gérer les sources d'interruptions des périphériques, le contrôleur OpenPIC comporte des mécanismes de communication inter-coeur et des minuteries globales. Ces deux types de fonctions peuvent interrompre n'importe quelle combinaison de coeurs à la fois, ce qui permet la réalisation de mécanismes de synchronisation à partir d'évènements asynchrones. Toutes ces sources d'interruptions globales sont réservées et ne peuvent être employées que par des pilotes du noyau afin de simplifier les règles d'assignation des ressources. Notre implémentation multicoeur du noyau n'emploie aucune des fonctions globales du contrôleur OpenPIC. Cependant, ces dernières pourraient éventuellement être employées à travers des hyperappels pour simplifier la réalisation de la synchronisation multicoeur.

Un problème identifié dans l'architecture initiale de XtratuM est que les sources d'interruptions externes sont toujours activées, peu importe à qui elles appartiennent. Ainsi, une inter-

ruption appartenant à une partition P1 pourrait survenir pendant la fenêtre d'activation d'une partition P2. Le gestionnaire d'interruptions externes dans XtratuM enregistrerait alors l'occurrence de l'interruption dans un champ de bits de la partition à laquelle la source appartient, soit P1. Une exception virtuelle n'est exécutée que si l'interruption appartient à la partition active. Dans notre exemple, il n'y aurait pas d'exécution superflue d'une exception virtuelle. Cependant, l'exécution d'un gestionnaire d'interruption pour une source n'appartenant pas à la partition active P2 équivaut à une subtilisation de cycles imprévue à l'encontre des contraintes du partitionnement temporel robuste. Ce problème a déjà été identifié par Rushby [57] dans le contexte du partitionnement robuste et par Littlefield-Lawwill et Kinnan [41] dans le contexte de l'IMA. Notre solution consiste à modifier le changement de contexte de partitions pour inclure une reconfiguration du contrôleur d'interruptions. Ainsi, seules les sources d'interruptions appartenant à la partition active seront traitées et l'analyse d'ordonnancement peut être faite avec un nombre réduit de sources potentielles d'évènements asynchrones, au détriment d'un temps de latence d'interruption élevé pour les sources n'appartenant pas à la partition active. Dans plusieurs systèmes IMA, la seule interruption permise est celle de l'horloge [41]. Nous permettons l'utilisation d'interruptions dans notre adaptation, mais les partitions critiques doivent préférablement employer l'interrogation continue des périphériques (« polling ») au lieu des interruptions, car c'est un mécanisme directement analysable par les outils d'analyse statique et de détermination du WCET. Cependant, l'approche par « polling » dégrade considérablement le facteur de charge du processeur, en raison du temps perdu à l'attente active d'évènements.

4.7 Mécanisme d'hyperappels

Aucune adaptation particulière n'a été nécessaire au niveau du mécanisme d'hyperappels pour l'implémentation multicoeur du noyau. Cependant, quelques hyperappels ont été ajoutés ou modifiés, afin de supporter l'adaptation multicoeur. De plus, la convention d'appel des hyper-

appels doit être adaptée au processeur ciblé, mais ces adaptations n'affectent pas le traitement des hyperappels en tant que tel.

4.8 Partitionnement temporel

Le support de l'exécution parallèle de plusieurs partitions est l'objectif principal de l'adaptation d'un noyau de partitionnement robuste vers une architecture multicoeur. Les systèmes IMA actuellement déployés sont pour la plupart des systèmes distribués qui exécutent des partitions communiquant entre elles sur plusieurs machines interconnectées [68, 55, 35] par des bus temps réel sûrs tels que ARINC-659 (SAFEBus) et ARINC-664 (AFDX). D'ailleurs, Rushby traite autant du partitionnement monoprocesseur que des répercussions du partitionnement distribué dans son rapport séminal sur le partitionnement robuste [57]. L'exécution parallèle de partitions dans un système n'est donc pas une nouvelle idée.

La différence majeure entre le partitionnement robuste distribué et le partitionnement robuste sur processeurs multicoeurs est le débit de communication atteignable entre les partitions. Dans les systèmes distribués, le débit globalement atteignable entre les partitions est limité à un ordre de grandeur d'environ 10 Mo/s dans le meilleur des cas [11] pour l'ensemble des communications. Certaines contraintes de synchronisation globales doivent être respectées, mais la quantité de données partagées et l'architecture des bus de communications permettent des latences tolérables de l'ordre des centaines de microsecondes. Dans les processeurs multicoeurs, la communication entre les partitions peut s'effectuer à travers des bus internes à très haute vitesse. Il devient ainsi possible de communiquer localement des données à un débit plusieurs ordres de grandeur supérieurs à celui des bus de communications inter-machine traditionnels. En particulier, le débit est assez élevé pour permettre l'utilisation du parallélisme au niveau des tâches (« task-level parallelism », en anglais) comme dans les systèmes d'exploitation traditionnels.

L'exploitation du parallélisme exposé par les processeurs multicoeurs requiert cependant l'adaptation des modèles de partitionnement robuste existants. En effet, l'unité de séparation des fonctions demeure la partition et les partitions existent pour permettre de faire respecter l'isolation entre les fonctions. Il faut spécifier des modifications au modèle de partitionnement robuste qui permettront d'exploiter le parallélisme sans mettre en péril l'isolation nécessaire entre les partitions.

Il n'existe actuellement aucune norme pour le partitionnement robuste multicoeur. La norme ARINC-653 est explicitement une norme vouée au partitionnement d'un seul processeur doté d'un seul coeur. Cependant, il existe différents modèles avancés par l'industrie pour s'attaquer au problème. Les deux modèles les plus avancés à notre connaissance sont ceux de SYSGO pour PikeOS [30] et ceux d'une équipe de Honeywell publié sous la forme d'un brevet [65].

Pour l'adaptation multicoeur de XtratuM, nous avons choisi d'étendre le modèle existant afin qu'il supporte plusieurs partitions et différents types de plans d'exécution. Le modèle original de XtratuM est basé sur ARINC-653 avec quelques différences. Malgré que la norme ARINC-653 et le modèle de partitionnement original de XtratuM ne mentionnent aucunement le partitionnement parallèle, les extensions nécessaires se sont avérées assez minimes.

L'élément central du partitionnement multicoeur est la composante temporelle : on veut pouvoir exécuter plusieurs partitions en même temps sur un même processeur. Les extensions au partitionnement spatial découlent ensuite directement de l'implémentation du partitionnement temporel parallèle.

4.8.1 Plans d'exécution multicoeur

Puisque nous désirons permettre l'évaluation de plusieurs types de plans d'exécution parallèles, nous avons adapté les composantes de partitionnement temporel de XtratuM afin d'être maximalement flexibles. Nous présentons ici les types de plans d'exécution multicoeur supportés.

4.8.1.1 Modèle asymétrique (AMP)

Le modèle AMP sur N coeurs est équivalent à exploiter N machines indépendantes avec chacune un noyau de partitionnement robuste. La puissance de calcul rendue disponible avec les coeurs additionnels n'est utilisable que pour augmenter le nombre d'applications intégrées sur la plateforme, mais pas pour exploiter le parallélisme directement.

La figure 4.3 présente un exemple de plan d'exécution asymétrique sur trois coeurs. On observe dans la figure que chaque partition n'est assignée à des tranches de temps que sur un seul des coeurs. Dans cet exemple, le coeur 1 est assigné à la partition B. De plus, la partition critique C est exécutée en exclusivité dans une tranche de temps durant laquelle les autres coeurs sont au repos afin d'éviter toute interférence causée par les ressources partagées avec les autres coeurs. Dans le cas asymétrique, l'utilisation d'un bloc d'activation majeur de périodicité identique sur chaque coeur n'est pas strictement nécessaire. Cependant, notre exemple emploie cette restriction par souci de simplicité.

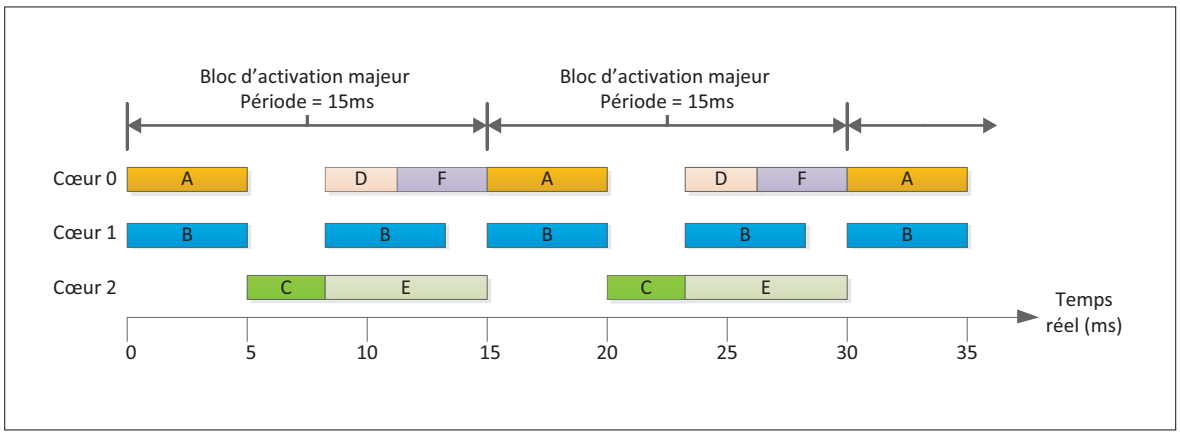


Figure 4.3 Plan d'exécution asymétrique (AMP) sur trois coeurs

Dans les plans d'exécution asymétriques, une partition ne peut résider que sur un seul coeur et ne peut pas migrer vers un autre coeur.

4.8.1.2 Modèle symétrique (SMP)

Le modèle SMP pur permet à chaque partition d'exploiter tous les coeurs en parallèle. Chaque partition peut donc exploiter au maximum le parallélisme offert par le processeur.

La figure 4.4 présente un exemple de plan d'exécution symétrique sur trois coeurs. On observe dans cet exemple trois groupes de partitions qui exploitent chacun les trois coeurs. Ces groupes de partitions exécutent chacun une même image de programme, associée à une même application, et doivent se synchroniser entre elles à l'aide d'opérations de synchronisations offertes par le noyau. Chaque groupe de partitions peut être considéré comme une « partition multicoeur logique » qui exploite un processeur multicoeur virtuel. Les développeurs de partitions sont libres d'exploiter ce parallélisme avec le niveau de concurrence désiré. Par exemple, un seul fil d'exécution pourrait s'exécuter sur chaque coeur, ou encore un système d'exploitation avec ordonnancement multicoeur avec plusieurs fils d'exécution (tâches) pourrait être utilisé. La communication entre les partitions d'un même groupe, par exemple entre A0, A1 et A2, qui partagent la même fenêtre d'activation, s'effectue à l'aide de mémoire partagée. L'utilisation de mémoire partagée pour la communication entre les partitions d'un groupe permet d'augmenter les performances lorsqu'on emploie un modèle de programmation parallèle au niveau des tâches (« multi-threaded » en anglais). Cependant, cette utilisation de mémoire partagée doit être prise en compte dans le modèle de partitionnement spatial. L'exploitation directe du parallélisme offert par les processeurs multicoeurs augmente donc le couplage entre les mécanismes de partitionnement spatial et temporel. Nous présentons les ajustements nécessaires à la section 4.9.2.

Dans les plans d'exécution symétriques, les règles suivantes s'appliquent :

- chaque partition est assignée à un groupe parallèle avec ses pairs (ex : A0, A1, A2 dans la figure 4.4) ;

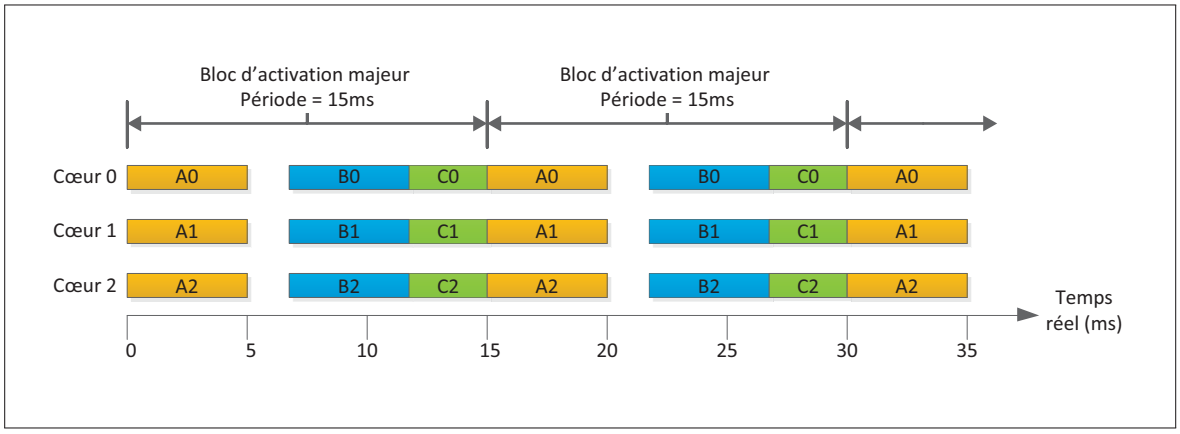


Figure 4.4 Plan d'exécution symétrique (SMP) sur trois coeurs

- la communication inter-partitions s'effectue directement par mémoire partagée, sans passer par le noyau ;
- la synchronisation entre les coeurs doit être réalisée explicitement par le système d'exploitation de partition qui s'exécute sur le groupe ;
- les partitions d'un même groupe doivent partager une image de programme unique.

4.8.1.3 Modèle hybride

Le modèle hybride est la généralisation des modèles AMP et SMP. Ce modèle exécute des groupes de partitions SMP qui exploitent de 1 à N coeurs dans une fenêtre d'activation. Il n'est pas nécessaire pour les groupes de partitions SMP d'exploiter tous les coeurs à la fois. Les partitions n'exploitant pas le parallélisme sont simplement décrites comme des groupes de partitions à un seul coeur.

La figure 4.5 présente un exemple de plan d'exécution hybride sur trois coeurs. Dans cet exemple, le groupe de partitions B1-B2 est un groupe SMP à deux coeurs. Toutes les autres partitions sont séquentielles et n'exploitent pas directement le parallélisme offert par la machine. La partition C demeure une partition critique qui est exécutée dans une fenêtre d'activation exclusive sans interférence possible par les autres coeurs. Ce plan d'exécution offre

la plus grande flexibilité et démontre l'exploitation maximale du processeur par des partitions non critiques alors qu'une partition critique se voit affecter une fenêtre d'activation maximalement déterministe équivalente à un système monocoeur.

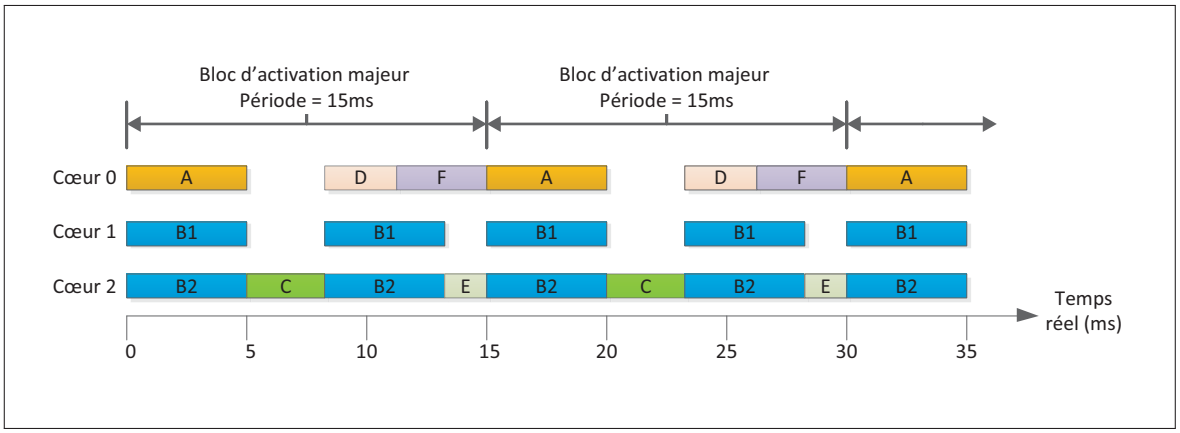


Figure 4.5 Plan d'exécution hybride sur trois coeurs

4.8.2 Ordonnancement multicoeur

Après avoir couvert les différents modèles de plans d'exécution possibles, nous pouvons présenter l'adaptation du noyau XtratuM qui les rend possibles. La différence entre les modèles SMP et AMP est uniquement conceptuelle. Les partitions d'un même groupe demeurent indépendantes, mais certaines règles de partitionnement spatial sont relaxées afin de permettre la communication par mémoire partagée. De plus, certains hyperappels de gestion du processeur multicoeur virtuel se servent d'attributs qui sont uniquement ajoutés aux partitions d'un groupe SMP pour déterminer si l'opération est légale dans ce contexte.

Il peut sembler a priori que des modifications majeures à l'ordonnanceur statique monocoeur de XtratuM soient nécessaires pour permettre un ordonnancement parallèle des partitions. Cependant, une décision-clé de l'adaptation du noyau a permis de conserver la totalité du code original d'ordonnancement de XtratuM. Cette décision est l'exécution SMP du noyau.

Comme nous l'avons présenté à la section 4.6.1, chaque coeur exécute une copie identique du noyau, mais dans un contexte local. Chaque instance locale du noyau exécute un plan d'exécution qui lui est propre et gère son MMU selon les partitions qui lui appartiennent. La seule différence entre un groupe SMP de partitions parallèles et une partition séquentielle est que le groupe SMP peut partager son espace mémoire avec ses pairs durant leur fenêtre d'activation commune. Puisque ce partage d'espace mémoire est géré par la composante spatiale du noyau de partitionnement XtratuM, l'ordonnanceur n'est aucunement affecté par la construction du plan d'exécution parallèle. L'ordonnanceur statique original de XtratuM est donc exécuté séparément sur chaque coeur et fait respecter le plan d'exécution assigné à chaque coeur dans le PCS sans avoir connaissance des relations de parallélisme entre les partitions.

Alors que l'ordonnanceur statique du noyau de partitionnement robuste n'est aucunement affecté par les considérations de parallélisme entre les partitions, il en est tout autrement pour l'ordonnanceur de deuxième niveau qui peut être réalisé dans le système d'exploitation de partition.

Dans le cas d'un groupe de partitions parallèle exécuté avec un seul fil par coeur, sans système d'exploitation de partition, le modèle de programmation est de bas niveau et les opérations de synchronisation doivent être directement employées entre les coeurs virtuels. Cette approche a l'avantage de la simplicité, mais elle est aussi très peu flexible puisque les développeurs sont limités à un seul fil d'exécution par coeur.

Une plus grande flexibilité d'exploitation du parallélisme par un groupe de partitions parallèle peut être obtenue en développant ou en adaptant un système d'exploitation de partition supportant un ordonnancement temps réel SMP. Notre adaptation multicoeur du noyau XtratuM virtualise la machine pour permettre l'exploitation des coeurs disponibles à un groupe de partitions de manière transparente. En particulier, les opérations nécessaires au démarrage multicoeur et à l'identification des différents coeurs sont disponibles sous forme d'hyperappels

(voir la section suivante). Il en revient aux utilisateurs potentiels de déterminer les problèmes de sûreté potentiels découlant des différents systèmes d'exploitation de partition SMP dans leur application. L'analyse plus poussée de ces considérations déborde du cadre du présent mémoire.

4.8.3 Hyperappels de support multicoeur

Les plans d'exécution SMP et hybrides présentés auparavant permettent d'utiliser plusieurs coeurs à la fois dans un groupe de partitions afin de profiter du parallélisme disponible dans un processeur multicoeur.

La partition principale d'un groupe de partitions est l'équivalent du coeur principal (coeur 0) dans un processeur réel. Il doit être possible pour la partition principale de démarrer virtuellement les autres partitions du groupe sur les autres coeurs, afin d'émuler un modèle SMP.

Notre adaptation multicoeur de XtratuM comporte de nouveaux services accessibles par hyperappels d'opérations abstraites de gestion des coeurs. Puisque les groupes de partitions SMP sont composés de partitions, une par coeur, les opérations existantes de gestion des partitions peuvent être utilisées pour gérer le démarrage, le redémarrage ou la suspension des autres coeurs.

Les hyperappels de gestion des partitions `XM_*_partition` ont été adaptés afin de leur permettre de cibler toutes les partitions du groupe SMP de la partition appelante. Par exemple, pour démarrer les autres coeurs à partir de la partition principale d'un groupe, on utilise l'hyperappel `XM_reset_partition` avec l'identifiant de la partition sur le coeur désiré. Les versions originales de ces hyperappels requéraient le statut superviseur, qui est un niveau de privilège trop élevé pour les applications générales.

En ce qui a trait à l'identification des coeurs, nous avons ajouté trois hyperappels qui permettent d'identifier les processeurs faisant partie d'un groupe de partitions SMP pour permettre le démarrage ordonné des partitions d'un groupe. Ces hyperappels sont les suivants :

- *XM_smp_is_master() :* Retourne une valeur booléenne vraie si la partition appelante est la partition principale d'un groupe.
- *XM_smp_get_cpu_id() :* Retourne l'identifiant logique de coeur sur lequel est exécutée la partition appelante.
- *XM_smp_get_peer_cpus() :* Retourne un champ de bits permettant d'identifier quels coeurs du processeur réel sont utilisés par le groupe de la partition appelante.

L'implémentation de ces adaptations a nécessité la modification de plusieurs fonctions utilitaires du noyau utilisées par les hyperappels afin de corriger leur sémantique dans le cas multicoeur.

4.8.4 Problèmes liés au partitionnement temporel multicoeur

La majorité des problèmes de sûreté potentiels liés au partitionnement robuste multicoeur ont été relevés dans la composante temporelle de l'isolation. Une des caractéristiques les plus importantes pour simplifier la validation d'un système partitionné est le déterminisme. Autant le déterminisme temporel de la plateforme cible que le déterminisme logique de l'ordonnancement des étapes d'un programme sont importants pour assurer la validité des analyses de respect des échéances et la sémantique des programmes. Notre adaptation architecturale de XtratuM avec support des plans d'exécution multicoeur est déterministe puisqu'elle est basée sur une configuration statique et tous les algorithmes employés sont à délai d'exécution borné. Cependant, le déterminisme de l'implémentation peut être affecté par plusieurs caractéristiques du matériel sous-jacent. Nous présentons dans les prochaines sous-sections les problèmes potentiels que nous avons relevés qui pourraient avoir une incidence majeure sur l'effort d'analyse et de validation de notre modèle de partitionnement temporel multicoeur.

4.8.4.1 Analyse du délai maximal d'exécution des tâches

L'analyse du délai maximal d'exécution des tâches (« Worst Case Execution Time » ou WCET en anglais) est une étape nécessaire dans la validation des plans d'exécution et d'ordonnancement. Dans le noyau de partitionnement robuste, le WCET de chaque hyperappel doit être fourni aux équipes qui développent les partitions afin de supporter leurs propres analyses de WCET. Dans les partitions, le WCET de chaque tâche exécutée par un ordonnanceur de système d'exploitation de partition doit aussi être connu. Ces WCET sont utilisés, entre autres, pour assurer que la fenêtre d'activation des partitions est suffisante et pour attribuer les priorités aux différentes tâches.

Les outils robustes existants pour l'analyse statique du WCET sont limités aux programmes séquentiels [30]. Des progrès sont réalisés à l'heure actuelle dans l'évaluation statique et dynamique du WCET pour les programmes parallèles exécutés sur plusieurs coeurs ou processeurs simultanément [51]. Cependant, les caractéristiques matérielles des processeurs multicoeurs du domaine commercial, dont ceux de l'architecture PowerPC, rendent ces analyses très difficiles. En effet, le partage des caches, les bus d'accès aux ressources partagées, les pipelines complexes et l'exécution superscalaire sont tous des obstacles à l'analyse fiable du WCET [37, 30, 21, 42].

Avec les outils actuels, tels que Rapitime de Rapita Systems et aiT de AbsInt, plusieurs restrictions sur le modèle de programmation doivent être respectées pour obtenir des valeurs de WCET fiables [21]. L'utilisation symétrique de plusieurs coeurs, tel que c'est le cas avec les modèles SMP ou hybrides, n'est pas supportée directement par ces outils sur les processeurs PowerPC.

Les travaux récents du projet MERASA [64] ont mené à une architecture multicoeur propice à l'analyse fiable du WCET, mais celle-ci n'en est qu'à l'état de prototype.

La solution proposée par Fuschen [30] est de forcer l'exécution monocoeur exclusive de toutes les partitions ayant des contraintes temporelles critiques qui pourraient être affectées par les partages de ressources entre coeurs, comme nous l'avons montré dans le cas de la partition « C » dans les figures 4.3 et 4.5. Il est alors possible de faire l'analyse classique de WCET à l'aide d'outils existants puisque l'ordonnanceur statique de partitions du noyau force le respect des tranches d'activation, garantissant ainsi le respect des contraintes nécessaires aux outils. Le désavantage majeur de cette approche est une sous-utilisation des ressources disponibles. De plus, en suivant cette approche, on proscrit l'utilisation de plusieurs coeurs aux applications critiques, alors que l'amélioration des performances de cette classe d'application est aussi importante que dans le cas des applications non critiques. La disponibilité d'un noyau comme XtratuM-PPC permettrait à d'autres chercheurs de faire progresser les recherches sur la mitigation de ce problème sur l'architecture PowerPC.

4.8.4.2 Synchronisation temporelle des ordonnanceurs

Comme nous l'avons mentionné, les groupes de partitions SMP n'ont aucun statut particulier au niveau de l'ordonnancement. Puisque l'ordonnanceur local sur chaque coeur est exécuté indépendamment, il est possible que de légers décalages temporels entre les coeurs apparaissent comme des temps morts durant lesquels certains des coeurs ne sont pas disponibles. Cet effet est observable à deux moments : au début et à la fin d'une fenêtre d'activation. Les décalages temporels élevés qui pourraient causer cette catégorie de problèmes proviennent de l'erreur de synchronisation globale des horloges.

La figure 4.6 illustre l'effet du décalage à l'aide d'un plan d'exécution SMP avec des décalages temporels exagérés entre les coeurs. Dans ce plan d'exécution, les partitions A0, A1 et A2 auraient dû être activées et désactivées à des moments identiques. La différence entre le moment de désactivation cause des temps morts dans A0 et A1 qui sont perçus par A2 comme des coeurs absents.

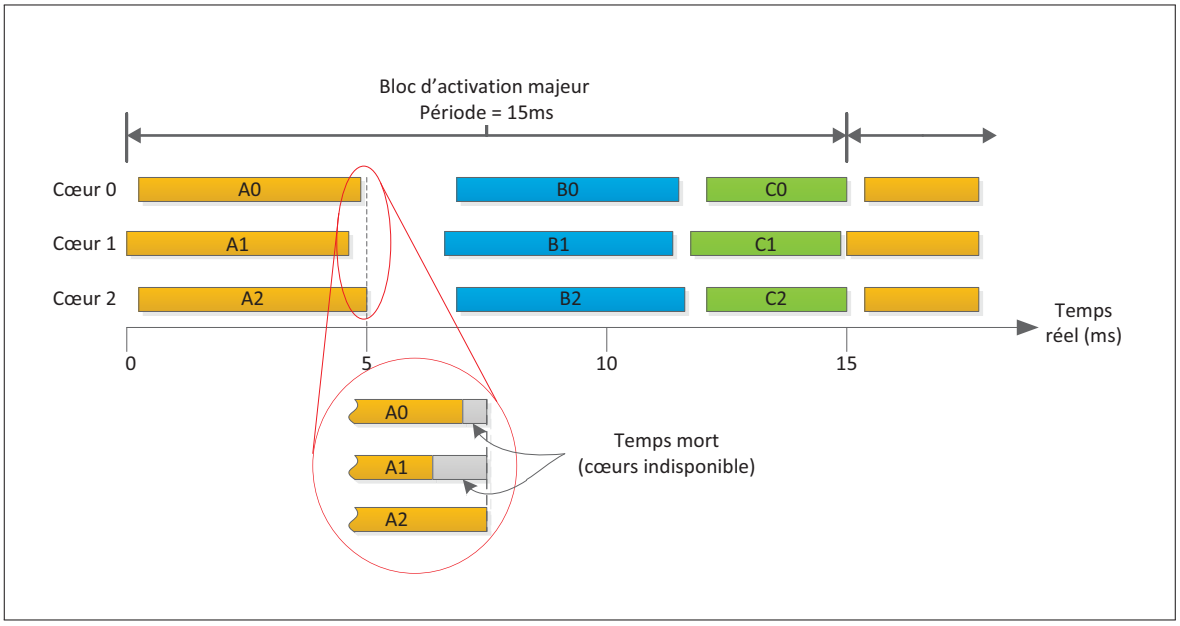


Figure 4.6 Temps morts causés par le décalage temporel entre ordonnanceurs locaux

Une conséquence du temps mort pourrait être la subtilisation de temps inopinée. Par exemple, si la partition A1 avait obtenu un verrou avant d'être désactivée et que la partition A2 voulait accéder à la même région critique, elle perdrait le reste de son temps de la fenêtre d'activation, ce qui pourrait totaliser des centaines, voire milliers de cycles. Ce cas serait problématique si la partition A2 avait été programmée en assumant que le verrou serait toujours relâché très rapidement, avant la fin de la fenêtre d'activation.

Une autre possibilité est l'apparence de délais plus élevés que prévu si des protocoles de synchronisation assument que tous les coeurs virtuels sont disponibles en même temps à tout moment.

Une solution potentielle à ce problème consiste à modifier les ordonnanceurs pour être activés par une interruption provenant d'une minuterie globale au lieu d'une minuterie locale du coeur. Cette solution permettrait de régler le problème, mais forcerait l'utilisation de tranches d'activation parallèles toujours égales entre les coeurs, ce qui réduirait la flexibilité de la construction des plans d'exécution. Cette solution pourrait être évaluée par d'autres chercheurs dans

une version ultérieure du noyau. Une approche alternative est d'employer la programmation défensive et des mécanismes de mitigation au niveau des programmes dans les partitions, en sachant que cet effet pourrait être observé. En pratique, les applications IMA sont souvent conçues afin de tolérer une certaine gigue de communication en raison des types de réseaux avioniques actuellement déployés qui présentent des gigues pouvant aller jusqu'à 500μs [11]. Les mécanismes de mitigation nécessaires sont donc bien connus par les praticiens du domaine.

4.8.4.3 Gestion des fautes

Quelle action le noyau de partitionnement robuste devrait-il exécuter dans le cas où une faute se produit dans une partition d'un groupe SMP ? Cette question met au jour un problème majeur qui se présente dans les plans d'exécution avec groupes de partitions parallèles.

Prenons pour exemple un débordement de tampon qui cause une faute d'accès mémoire dans une partition A. Cet exemple est illustré par le plan d'exécution de la figure 4.7. L'action programmée dans le PCS est la suspension de la partition par le moniteur de santé du système (HM dans la figure). Une partition superviseur nommée « IVHM » est programmée pour redémarrer la partition fautive si les conditions le permettent. La partition IVHM est exécutée dans une fenêtre d'activation ultérieure au moment de la faute.

Le cas monocoeur séquentiel est présenté dans la partie supérieure de la figure 4.7. Dans ce cas, la partition A est suspendue au moment de la faute et redémarrée ultérieurement par la partition IVHM. Aucune autre partition ne dépend de la partition A dans la même fenêtre d'activation.

Le cas multicoeur avec groupes de partitions parallèles est présenté dans la partie inférieure de la figure 4.7. Dans ce cas, avec les partitions A0, A1 et A2 en parallèle, que doit-on faire lors d'une défaillance de A0 ? Deux possibilités d'actions du HM se présentent : redémarrer ou suspendre la partition fautive. Plusieurs nouvelles questions se posent alors :

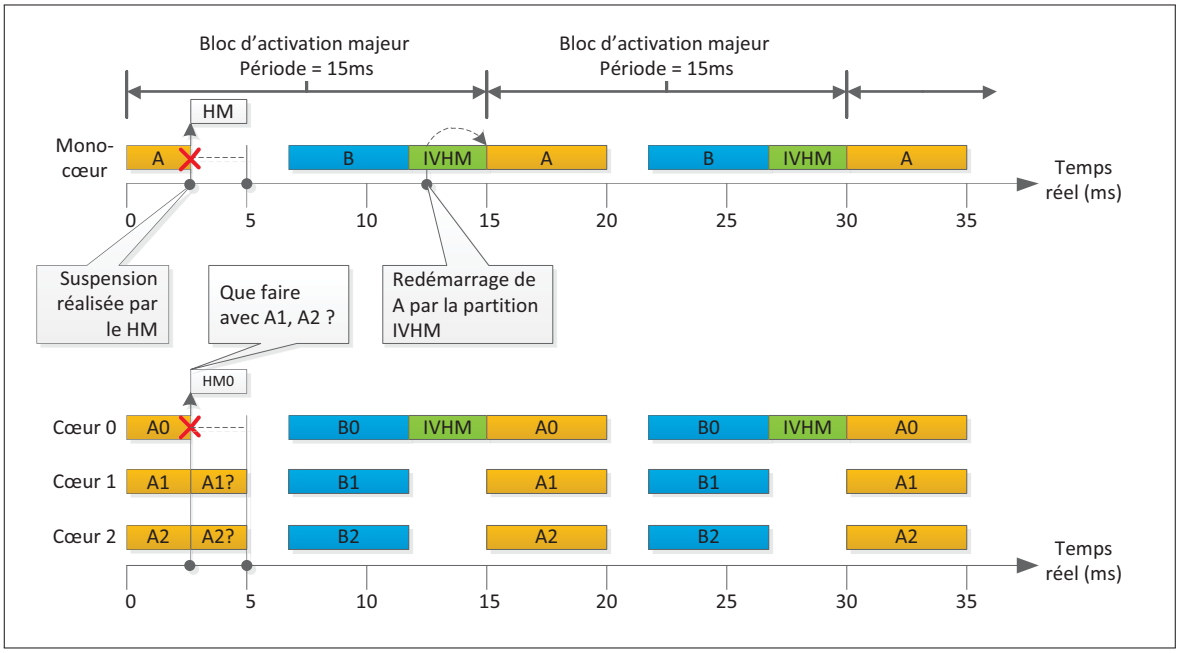


Figure 4.7 Exemple de gestion d'une faute dans un groupe de partitions SMP

- Est-ce que la faute pourrait avoir affecté des données nécessaires au fonctionnement correct de A1 et A2 ?
- Est-ce que le redémarrage de A0 dans le contexte du système d'exploitation de partition utilisé causerait des incohérences de synchronisation ou de données ?
- Est-ce que la suspension ou le redémarrage de A0 auraient comme effet l'arrêt de la progression des opérations de A1 et A2 ?

Les couplages de partages de ressources entre les partitions A0, A1 et A2 rendent l'analyse des défaillances plus complexe que dans le cas séquentiel. De plus, l'action à prendre est probablement conditionnelle à l'état actuel du traitement. Lorsqu'une faute est de nature à nécessiter une suspension ou un redémarrage de partition, il s'avère que le pire cas doit être pris en considération, au risque de permettre un effet domino de nouvelles fautes qui surviendraient en raison de la suspension ou du redémarrage partiel d'un groupe.

Des travaux additionnels devraient être effectués pour déterminer l'adaptation la plus adéquate des mécanismes actuels afin d'éviter la défaillance complète d'un groupe de partitions lors-

qu'une des partitions du groupe est fautive. Nous avons conservé les mécanismes actuels de moniteur de santé dans XtratuM, qui ne prennent aucunement en considération les groupes de partitions à défaut d'avoir une solution adéquate dans tous les cas.

Une autre version du problème de gestion des fautes survient lors d'une faute au niveau du noyau plutôt qu'au niveau d'une partition. Dans ce cas, si l'action du moniteur de santé requiert le redémarrage du coeur, est-ce que le système en entier devrait être redémarré ? La réponse est oui, car le démarrage du noyau d'un coeur implique une synchronisation de tous les coeurs (voir section 4.6.2). Dans le cas d'une telle faute, il n'est donc pas possible de continuer le traitement sur les autres coeurs ou de les utiliser pour tenter d'y remédier. De plus, une faute au niveau du noyau pourrait indiquer une corruption de données, ce qui implique le rechargement de l'image du noyau. Si le noyau doit être rechargé, il est impossible de continuer à l'exécuter sur les autres coeurs. Plus le nombre de coeurs est élevé, plus il risque d'y avoir une grande quantité d'applications exécutées sur la même machine afin de maximiser l'utilisation des ressources. Paradoxalement, cette meilleure intégration risque de causer des conséquences beaucoup plus graves lors d'une défaillance du noyau, car le redémarrage entier de la machine causerait la perte d'un plus grand nombre de fonctions que dans le cas des machines à processeurs monocoeurs. Ce grave problème est causé par l'exécution SMP du noyau. Si un noyau de partitionnement robuste indépendant, sans aucun couplage entre les coeurs, était exécuté sur chaque coeur, le redémarrage d'un coeur aurait un effet beaucoup moins grand. Cependant, l'exploitation maximale du parallélisme offert par le processeur serait limitée, car seuls les plans d'exécution purement AMP pourraient être employés. Il serait possible de modifier notre adaptation XtratuM pour permettre une telle exécution AMP, mais ce sont des travaux futurs qui n'ont pas été réalisés dans le cadre de nos travaux.

4.8.4.4 Modèles de programmation parallèle

Nous n'avons pas, jusqu'à présent, spécifié le modèle de programmation parallèle à exploiter par les groupes de partitions parallèles. Comme nous l'avons mentionné à la section 4.8.2, notre adaptation du noyau XtratuM devrait permettre l'exploitation de tous les modèles de programmation multicoeur puisque la mémoire partagée est disponible nativement, de même que les primitives de synchronisation et les opérations de gestion des coeurs.

Cependant, puisque nous n'avons pas directement évalué ces modèles de programmation, il est possible que certains des modèles de programmation parallèles traditionnels ne soient pas directement applicables sous XtratuM-PPC. La sûreté et le déterminisme des applications sur une plateforme IMA sont affectés par le noyau de partitionnement robuste, mais aussi par leur conception architecturale et par leur implémentation. Des travaux additionnels devraient être effectués pour adapter des systèmes d'exploitation SMP existants à XtratuM-PPC dans le but de déterminer les problèmes de sûreté potentiels à ce niveau.

4.9 Partitionnement spatial

L'adaptation multicoeur du partitionnement spatial dans XtratuM-PPC est l'élément-clé qui permet l'exploitation des différents types de plans d'exécution multicoeur présentés dans la section précédente. Nous avons conçu une extension au modèle de partitionnement spatial original permettant le partage de données entre partitions d'un groupe de partitions parallèles. De plus, nous avons ajouté un support complet de protection mémoire à l'aide d'un MMU. Finalement, nous avons ajouté des protections par verrou aux structures de données qui pourraient être accédées par plusieurs coeurs à la fois. Toutes ces adaptations font en sorte que les plans d'exécution multicoeur permis respectent les contraintes de partitionnement spatial et temporel.

4.9.1 Exploitation du MMU

Le support du MMU manquant dans la version monocoeur pour LEON de XtratuM était nécessaire afin de supporter un modèle de partitionnement spatial multicoeur. De plus, l'ajout de ce support comble une lacune importante du noyau original en ce qui a trait au respect d'une isolation forte entre les espaces mémoire des différentes partitions.

Puisque XtratuM se veut un noyau de partitionnement pour applications temps réel, nous avons proscrit les fautes de pages et la recherche de table de pages aidées par le matériel. Ces deux fonctionnalités permettent d'exploiter un nombre de pages et de blocs de translation mémoire supérieur à ce qui est disponible dans la cache du MMU (TLB ou « Translation Lookaside Buffer », en anglais). Elles réduisent cependant le déterminisme du temps d'exécution du code, car des exceptions dépendantes des flux de données et d'instructions peuvent survenir à tout moment. La translation des adresses par le MMU est donc limitée à une configuration statique des TLB dans XtratuM-PPC. Étant donné que les TLB enregistrent des mappages directs entre les adresses virtuelles et réelles, les délais d'accès sont constants et les fautes de pages ne surviennent que lorsqu'une erreur de protection mémoire survient.

Nous définissons deux types de ressources de MMU : les pages à translation directe (PTD) et les blocs à translation directe (BTD). La différence entre les PTD et les BTD se trouve au niveau de la taille : les pages sont de tailles fixes et sont habituellement entre 4 ko et 64 ko dans la plupart des architectures. Les blocs sont de tailles fixes ou variables et peuvent habituellement couvrir des plages beaucoup plus grandes que les pages. La dénomination « à translation directe » est utilisée pour rappeler qu'il s'agit de ressources disponibles en TLB. Le nombre de PTD et de BTD disponibles dépend de l'architecture. Par exemple, dans les PowerPC e600, le TLB supporte 128 PTD de 4 ko pour les instructions et pour les données, ainsi que 8 BTD de 128 ko à 4 Go. Pour les PowerPC e500v2, le TLB supporte 512 PTD unifiées (données ou instructions) de 4 ko et 16 BTD unifiés de 4 ko à 4 Go.

L'utilisation de PTD et BTD pour le noyau et les partitions impose des restrictions de tailles. La taille maximale en mémoire d'une partition est bornée par la disponibilité d'une combinaison de PTD et de BTD couvrant l'espace désiré. Un certain nombre de PTD et BTD sont réservés en permanence pour le noyau. Le reste est disponible pour les partitions. Cela ne s'avère pas un problème en pratique, car les applications temps réel n'exploitent jamais un espace mémoire supérieur à la mémoire physique disponible, car la permutation de pages entre la mémoire et le disque (« swapping » en anglais) est une opération dont le délai d'exécution est difficile à borner. Le nombre de PTD et BTD est toujours suffisant pour couvrir l'ensemble de la mémoire physique dans les processeurs supportés par XtratuM-PPC. Il peut cependant y avoir des allocations de mémoire physique sous-optimales en raison du nombre fini d'entrées et de leur granularité.

À chaque changement de contexte de partition, les TLB sont vidangés et rechargés avec une table de PTD et BTD correspondant à la partition activée, donnant ainsi au processeur une vue de la mémoire limitée à l'espace alloué à cette partition dans le PCS. Les permissions assignées à chaque PTD et BTD sont configurées afin de forcer une exception du MMU si un accès interdit se produit. L'assignation des ressources du MMU aux partitions est décrite dans la section suivante. Les détails d'implémentation du changement de contexte sur PowerPC sont présentés à la section 5.9.

4.9.2 Adaptation multicoeur du modèle de partitionnement spatial

Les modèles de programmation parallèle à mémoire partagée utilisent des variables partagées entre les coeurs pour communiquer. Ces variables partagées sont accédées par les fils d'exécutions à la même adresse physique, en alternance. Des protocoles de synchronisation sont utilisés pour éviter l'accès non atomique à ces zones de mémoire partagée.

Par définition, il est interdit de partager des zones de mémoire entre les différentes partitions exploitées par un noyau de partitionnement robuste. Cette restriction améliore la sûreté dans

les systèmes monocoeurs, mais elle va directement à l'encontre des modèles de programmation parallèle à mémoire partagée. Nous avons adapté le modèle de partitionnement spatial de XtratuM de manière à ne permettre le partage de mémoire qu'à des partitions faisant partie d'un même groupe. Dans notre modèle, on considère les partages de mémoire entre les groupes au lieu d'entre les partitions.

Nous préciserons maintenant la construction et l'utilisation des groupes de partitions que nous avons mentionnées à quelques reprises dans les derniers chapitres. Le groupe de partitions est le mécanisme principal développé pour permettre l'exploitation du parallélisme tout en respectant l'esprit des contraintes d'isolation spatiale entre les partitions dans un noyau de partitionnement robuste.

Un groupe de partitions est un ensemble logique de partitions qui partagent des zones de mémoire et qui s'exécutent sur des coeurs mutuellement exclusifs. Les partitions membres d'un groupe parallèle sont définies de la même manière que les partitions séquentielles. En fait, les partitions séquentielles sont chacune allouées à un groupe ne contenant qu'une seule partition. L'image d'une partition contient le code (segment « .text ») ainsi que les données globales (segments « .bss » et « .data ») du binaire de la partition. Toutes les partitions d'un groupe doivent partager la même image de partition, qui est chargée dans un espace d'adresse physique commun. Cependant, chaque partition possède sa propre pile locale. En plus de l'image de partition, des zones de données partagées additionnelles peuvent être définies pour permettre la création de tampons alloués lors de l'initialisation des partitions.

Les groupes sont détectés implicitement à partir d'un ensemble de règles mises en oeuvre dans l'outil `xmcparser`. Lors de la compilation du PCS, `xmcparser` crée des groupes en fonction des assignations de zones de mémoire des partitions. Si les règles de regroupement sont respectées par plusieurs partitions assignées à des coeurs différents, les règles proscrivant les partages de mémoire entre les partitions sont relaxées pour les membres d'un même groupe.

Chacune des partitions doit être composée de zones de mémoire virtuelle appartenant à des zones de mémoire physique dans le PCS. Les attributs d'une zone de mémoire virtuelle déterminent son type et ses permissions d'accès. Les adresses virtuelles et réelles (physiques) de toutes les zones de mémoire virtuelle doivent être identiques. Ces restrictions sont imposées afin de faciliter la validation du PCS et de simplifier le débogage des applications.

Tableau 4.1 Types de zones de mémoire virtuelle dans XtratuM-PPC

Type	**Caractéristiques**
« code »	Utilisé pour le code exécutable en lecture seule.
« io »	Utilisé pour les zones d'entrées/sorties avec dérivation des caches.
« data »	Utilisé pour les données en lecture/écriture.
« stack »	Utilisé pour les piles.

Nous avons développé quatre types de zones de mémoire virtuelle, dont les caractéristiques sont présentées au tableau 4.1. Chaque type correspond à une configuration différente du PTD ou du BTD associé dans le MMU. Ces types de zones de mémoire sont utilisés pour la détection des groupes de partitions parallèles dans `xmcparser`. Ils sont aussi utilisés dans le noyau pour la configuration du MMU lors du chargement des partitions.

Une image de partition, peu importe qu'elle fasse partie d'un groupe parallèle, doit respecter les règles suivantes :

1. Elle doit posséder au moins une zone « code », une zone « data » et une zone « stack ».
2. Ses zones virtuelles doivent être assignées à une même région de mémoire physique.
3. Les zones de type « code », « data » et « stack » doivent se succéder respectivement dans cet ordre.

Lorsque l'outil `xmcparser` détecte plusieurs partitions sur des coeurs différents qui partagent la même image de partition, à l'exception des zones de pile (« stack »), ces dernières sont assignées à un même groupe. Lors de la validation des contraintes de partitionnement

par `xmcparser`, tels que le chevauchement de zones de mémoire, les partitions d'un même groupe peuvent partager des zones de type « code » et « data » librement sans produire d'erreur.

Pour mieux illustrer l'implémentation du partitionnement spatial multicoeur, nous avons construit un exemple à deux coeurs représentatif de partitions d'un groupe parallèle. Dans cet exemple, deux partitions font partie d'un groupe qui possède les caractéristiques suivantes :

- une image de partition de 512 ko, composée de :
  - une zone partagée de type « code » composée d'un BTD de 256 ko ;
  - une zone partagée de type « data » composée d'un BTD de 128 ko ;
  - une zone de pile (type « stack ») composée de deux PTD de 4 ko pour chaque coeur.
- une région additionnelle de données de 512 ko composée d'un BTD de 512 ko.

La carte mémoire de cet exemple est présentée à la figure 4.8. Dans cet exemple, toutes les partitions faisant partie du groupe parallèle partagent une même région « code » et « data ». Les piles sont cependant uniques à chaque coeur et sont identifiées dans la figure par « Pile CPU0 » et « Pile CPU1 ». La zone de données additionnelles de 512 ko est aussi partagée par toutes les partitions du groupe et pourrait servir à entreposer des données de traitement. On remarque que les règles énoncées précédemment pour les groupes de partitions parallèles sont respectées.

Comme dans le cas du partitionnement temporel multicoeur, le partitionnement spatial est réalisé localement par l'instance du noyau sur chaque coeur. Les fautes d'accès mémoire sont détectées localement par le MMU de chaque coeur. Le module de partitionnement spatial du noyau ne dépend pas de caractéristiques particulières des processeurs multicoeurs. Son implémentation est entièrement locale et séquentielle, sans partage de données entre les coeurs. Outre la présence d'un support complet de protection mémoire par MMU, il n'y a aucune différence logique au niveau du noyau entre la version originale monocoeur de XtratuM et notre

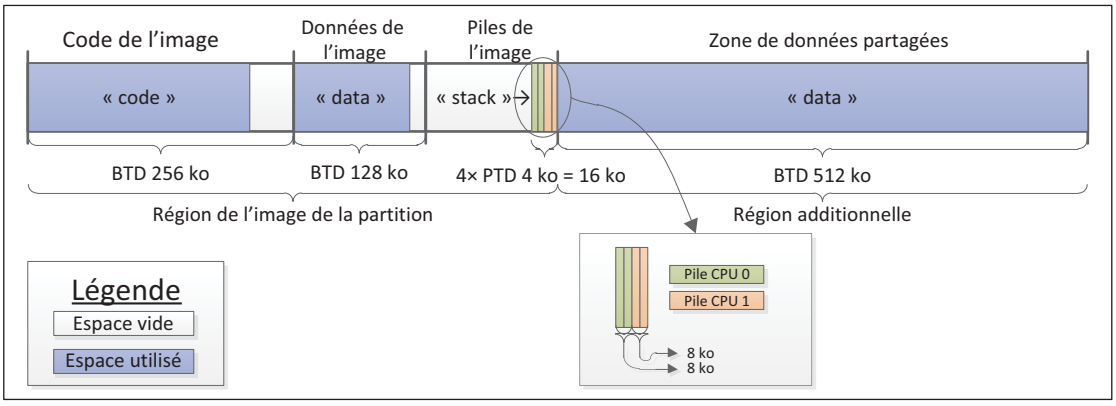


Figure 4.8 Exemple d'allocation de zones de mémoire virtuelle
pour un groupe de partitions parallèles sur deux coeurs

version multicoeur. Le reste des caractéristiques qui permettent l'application d'un modèle de partitionnement spatial multicoeur découlent entièrement de conventions appliquées par l'outil `xmcparser` lors de la compilation et de la validation du plan de configuration du système. L'exécution d'une copie locale du noyau sur chaque coeur suffit pour permettre l'application de tous les types de plans d'exécution parallèles que nous désirons supporter.

Finalement, nous devons mentionner que notre version multicoeur avec support MMU de XtratuM ne permet pas l'exploitation directe de la mémoire virtuelle par le système d'exploitation de partition. Puisque nous n'avons pas virtualisé le support du MMU de sorte à le rendre disponible aux partitions, les systèmes d'exploitation de partitions sont limités à l'utilisation d'un modèle de mémoire réelle. L'implémentation dans l'hyperviseur d'un MMU virtuel, comme cela est fait par exemple dans Xen [14], aurait grandement complexifié l'implémentation alors qu'il ne s'agissait pas d'un besoin primaire. Plusieurs des cas d'utilisation de support MMU dans les systèmes d'exploitation, incluant la protection de la mémoire, peuvent cependant être émulés avec des allocations statiques de PTD et BTD dans le plan de configuration du système.

4.9.3 Protection des données partagées

Nous avons ajouté des protections par verrous aux structures de données qui pourraient être accédées par plusieurs coeurs à la fois. L'accès concurrent aux objets du noyau tels que les pilotes de périphériques et les ports de communication inter-partitions sont évité par l'insertion d'un verrou local sur chaque objet. Chaque objet peut être implémenté pour supporter une gestion des verrous spécifique. Ainsi, il est possible de restreindre le verrouillage au corps de chaque appel sur les objets, comme il est possible d'appliquer le verrouillage globalement entre un appel `open()` et un appel `close()`. Les hyperappels d'accès aux objets (voir section 3.7) retournent immédiatement avec un code d'erreur si le verrou appartient déjà à une autre partition, ce qui indique que la ressource est déjà en cours d'utilisation sur un autre coeur. Ces verrous ne sont jamais directement accédés par les utilisateurs.

Notons que la concurrence d'accès aux ressources pourrait être entièrement éliminée en ajoutant des contraintes de validation dans l'outil `xmcparser`. Il serait donc possible de garantir qu'aucune erreur de ressource occupée ne survienne lors de l'exécution d'un plan d'exécution, si cela s'avérait nécessaire afin de simplifier l'analyse d'ordonnancement des partitions.

4.9.4 Problèmes liés au partitionnement spatial multicoeur

Nous présentons dans cette section les problèmes potentiels que nous avons relevés au niveau du partitionnement spatial. Comme nous l'avons déjà mentionné, le partitionnement temporel et spatial sont fortement couplés. Certains problèmes découlant de l'adaptation multicoeur du partitionnement spatial peuvent causer des délais temporels qui affectent l'isolation temporelle. Nous avons relevé trois problèmes liés au modèle de partitionnement spatial multicoeur.

Le premier problème est lié à la gestion adéquate des partages de données permis au sein d'un groupe de partitions parallèle dans le plan de configuration du système. Comme nous l'avons décrit à la section 4.9.2, l'exploitation du parallélisme au niveau des tâches dans les plans

d'exécution SMP et hybrides est rendue possible par la disponibilité de zones de mémoire partagée. Cependant, il en revient aux programmeurs des partitions et des systèmes d'exploitation de partition parallèles de gérer correctement l'accès à la mémoire partagée. Seuls les objets du noyau accessibles par hyperappels sont automatiquement protégés contre les accès concurrents. Les autres partages de données au niveau des partitions doivent être protégés afin d'éviter les conditions de course de données et les erreurs logiques d'ordonnancement. Ainsi, l'utilisation adéquate d'opérations atomiques et de verrous, tels que ceux présentés plus loin à la section 5.6, est primordiale afin de synchroniser les partages de données. L'adaptation multicoeur de XtratuM ne garantit pas à elle seule la sûreté des programmes parallèles en général. Au contraire, elle met au jour les catégories de bogues propres aux programmes parallèles mal synchronisés, qui peuvent causer des défaillances dans les partitions et mettre en péril l'isolation spatiale nécessaire, par exemple, au sein d'un système d'exploitation multicoeur de partition.

Le deuxième problème est l'insuffisance des mécanismes de protection d'accès concurrents aux objets fournis par le noyau. Les verrous internes de chaque objet du noyau ne permettent pas d'éviter une séquence invalide d'opérations de lecture et d'écritures lorsque plusieurs coeurs essaient d'opérer sur un objet en même temps. Par exemple, l'écriture vers un pilote de console pourrait nécessiter une séquence d'opérations qui garantisse que les blocs complets écrits par chaque coeur se retrouvent en sortie ordonnés par lignes de caractères. Si plusieurs coeurs écrivent vers la console sans se synchroniser à un plus haut niveau, le noyau évite des états invalides du pilote par ses verrous internes, mais le résultat est tout de même un mélange de caractères impossible à déchiffrer, car les caractères seront affichés selon le mode « premier arrivé, premier servi », alors qu'il serait plus logique que des lignes complètes de chaque transaction soient entrelacées. Pour résoudre ce problème, les développeurs de partitions doivent implémenter des protocoles de synchronisation et d'ordonnancement explicites à un plus haut niveau que le noyau. Il est impossible pour le noyau de connaître au préalable les contraintes

logiques d'une application qui sont affectées par l'exécution parallèle sur plusieurs coeurs. Ce problème n'est pas intrinsèque au noyau. Il s'agit plutôt d'une conséquence au niveau du système découlant de l'apparition de groupes de partitions parallèles.

Enfin, les verrous internes des objets du noyau utilisent l'algorithme « spinlock », qui ne fournit pas de garanties d'accès équitable aux ressources. Il est donc possible d'arriver à des ordonnancements d'opérations critiques qui font en sorte qu'un coeur est toujours défavorisé pour l'accès au verrou d'un objet souffrant d'une forte contention. Dans ces conditions, le coeur défavorisé pourrait même ne jamais réussir à accéder à la ressource partagée. Cette condition d'arrêt de progression doit être prise en compte dans les protocoles de synchronisation de plus haut niveau. Une solution à ce problème est la réalisation d'un serveur qui gère l'accès à l'objet partagé au sein d'une partition. Chacune des tâches désirant accéder à la ressource partagée emploie un port de communication qui lui est préassigné dans le serveur. La politique d'accès à la ressource peut ensuite être gérée de manière équitable par des algorithmes du serveur d'accès. L'utilisation d'un port de communication indépendant pour chaque acteur évite la contention vers un port unique dont l'accès concurrent est inéquitable. Un autre avantage de cette méthode est qu'elle permet de rendre explicite les relations de partages de ressources pouvant survenir en parallèle. Une autre solution possible, lorsque ce problème survient entre plusieurs partitions, est de s'assurer que les partitions ayant des dépendances de communication ne se retrouvent jamais dans des fenêtres d'activation se chevauchant sur différents coeurs. Cette solution est cependant assez inflexible puisque les plans d'exécution parallèles comportant des partitions critiques ont un nombre réduit de fenêtres d'activation.

Les problèmes que nous avons relevés au niveau du partitionnement spatial demeurent non-résolus. Les pistes que nous suggérons pourraient être évaluées plus en profondeur par d'autres chercheurs en se basant sur nos travaux.

4.10 Communication inter-partitions

Outre la protection des données partagées dans les objets de ports de communication, aucune adaptation particulière n'a été nécessaire au niveau de la communication inter-partitions pour l'adaptation multicoeur. Cependant, la possibilité d'exploiter des ports de communication inter-partitions en parallèle n'a pas été étudiée. L'ajout de contraintes de validation additionnelles dans l'outil `xmcparser` permettrait cependant de proscrire par convention les cas jugés problématiques lors de travaux ultérieurs.

4.11 Moniteur de santé du système

Notre adaptation du noyau XtratuM pour les processeurs multicoeurs ne comporte aucune adaptation directement liée au traitement parallèle. Les seuls changements dans XtratuM-PPC sont l'ajout des définitions d'exceptions propres aux processeurs PowerPC et l'ajout d'exceptions liées à la protection mémoire par MMU.

Cependant, l'utilisation de plans d'exécution multicoeur et l'exécution SMP du noyau causent des problèmes sémantiques dans le modèle de gestion des défaillances du moniteur de santé du système de XtratuM. Les cas les plus importants ont déjà été couverts à la section 4.8.4.3. Rappelons que l'utilisation de groupes de partitions parallèles complexifie l'analyse des défaillances. De plus, il n'est pas encore clair à l'heure actuelle comment procéder avec les autres partitions d'un groupe de partitions parallèles lorsqu'une partition membre subit une défaillance. Notre adaptation multicoeur de XtratuM permet l'évaluation de modèles de gestion des défaillances dans les systèmes multicoeurs. Nous n'avons cependant pas réalisé de travaux dans cette direction dans le cadre de ce mémoire.

4.12 Récapitulation

Dans ce chapitre, nous avons présenté les adaptations architecturales génériques nécessaires pour l'exploitation de plusieurs coeurs d'un même processeur dans XtratuM. Notre adaptation

visait à minimiser les changements nécessaires au noyau et aux outils, tout en permettant l'exploitation de trois types de plans d'exécution multicoeurs : AMP, SMP et hybrides.

Le support des plans d'exécution parallèles pour le partitionnement temporel est réalisé par l'exécution symétrique (SMP) du noyau sur chaque coeur. La duplication des fils d'exécution du noyau pour chaque coeur est effectuée lors du démarrage. L'alignement des lignes du temps d'ordonnancement de chaque coeur est assuré par la synchronisation de leur horloge locale.

Le module de partitionnement spatial dans notre adaptation exploite l'unité de gestion de mémoire (MMU) de chaque coeur pour assurer une détection matérielle des fautes d'accès et des tentatives de bris de l'isolation spatiale. Nous avons développé un modèle de protection mémoire basé sur des groupes de partitions parallèles qui peuvent partager des zones de mémoire durant leur fenêtre d'activation. Les partages sont permis par la relaxation des règles d'isolation spatiale inter-partitions lorsque la construction d'un groupe de partitions parallèles respecte un ensemble de règles. Les groupes de partitions parallèles sont une extension du modèle de partitionnement spatial existant de XtratuM, réalisé par l'ajout de propriétés et de contraintes dans le plan de configuration du système.

Comme nous l'avons démontré au cours du chapitre, les adaptations architecturales sont bien circonscrites. Notre version multicoeur de XtratuM est entièrement cohérente avec la version monocoeur et seul un nombre réduit d'hyperappels additionnels a dû être ajouté.

La majeure partie des additions architecturales pour supporter le partitionnement robuste multicoeur a été réalisée au niveau de l'outil de validation et de compilation du PCS (`xmcparser`). Ces additions à `xmcparser` implémentent les extensions multicoeurs à l'aide de conventions dans le plan de configuration du système. Deux conséquences importantes résultent de cette implémentation par conventions :

1. Le code noyau est fonctionnellement identique, peu importe que l'on cible un système à un seul ou plusieurs coeurs. L'absence de différences dans le PCS et dans le noyau entre l'exécution monocoeur et multicoeur de notre version adaptée simplifie sa validation.
2. Différentes contraintes de modèles de partitionnement peuvent être évaluées avec XtratuM-PPC sans avoir à modifier le noyau. Tous les détails de ces modèles sont entièrement gérés par des contraintes appliquées par l'outil `xmcparser`. Puisque cet outil est une application classique, sans idiomes de programmation embarquée, ni de contraintes temporelles ou spatiales, il est beaucoup plus facile d'y intégrer des changements et algorithmes complexes pour supporter les modèles désirés.

Nous avons relevé plusieurs problèmes lors de l'adaptation architecturale de XtratuM. Certains de ces problèmes découlent directement du passage d'un modèle de partitionnement purement séquentiel à un modèle de partitionnement parallèle. D'autres problèmes proviennent des différences architecturales entre les processeurs monocoeurs et multicoeurs. Un résumé des problèmes identifiés est présenté au tableau 4.2 à la page suivante.

Le prochain chapitre poursuit le traitement du développement de XtratuM-PPC en présentant les spécificités techniques de l'adaptation lorsque l'architecture PowerPC est ciblée.

Tableau 4.2 Résumé des problèmes potentiels de l'adaptation architecturale de XtratuM pour le support multicoeur

Description	**Section**
Si une défaillance survient sur un coeur lors du démarrage, l'action à prendre est indéterminée. Des recherches additionnelles sont nécessaires.	4.6.2, p.85
Dans l'architecture initiale de XtratuM, les sources d'interruptions externes sont toujours activées, peu importe à quelle partition elles appartiennent. L'arrivée d'une interruption ne ciblant pas la partition présentement active pourrait voler du temps d'exécution à cette dernière. Dans notre adaptation, ce problème est résolu par la désactivation des sources d'interruptions n'appartenant pas à la partition active lors du changement de contexte de partition.	4.6.4, p.89
Il n'existe actuellement aucune norme pour le partitionnement robuste multicoeur. Les implémentations existantes, dont la nôtre, sont des extensions aux modèles séquentiels existants, réalisées au meilleur des connaissances de chaque entité.	4.8, p.92
Les outils robustes existants pour l'analyse statique du WCET sont limités en pratique aux programmes séquentiels. Cependant, le WCET des fonctions du noyau et des tâches des partitions est nécessaire pour l'analyse de faisabilité de l'ordonnancement multicoeur. Des efforts additionnels au niveau du développement de ce type d'outils sont encore nécessaires avant d'en arriver à une situation satisfaisante.	4.8.4.1, p.100

Description	Section
Dans notre adaptation, l'ordonnanceur local sur chaque coeur est exécuté indépendamment. Il est possible que les faibles décalages temporels entre les coeurs apparaissent comme des temps morts durant lesquels certains des coeurs ne sont pas disponibles, ce qui pose un problème logique. Nous proposons d'employer la programmation défensive au niveau des programmes comme méthode de mitigation.	4.8.4.2, p.101
L'action à prendre par le moniteur de santé du système lorsqu'une faute se produit dans une des partitions d'un groupe de partitions parallèles demeure une question ouverte.	4.8.4.3, p.103
Il est possible que certains des modèles de programmation parallèles traditionnels ne soient pas directement applicables avec notre adaptation multicoeur de XtratuM. L'évaluation de ces modèles de programmation débordait cependant du cadre de nos travaux.	4.8.4.4, p.106
Il en revient aux programmeurs des partitions et des systèmes d'exploitation de partition parallèles de gérer correctement l'accès à la mémoire partagée offerte par notre modèle de partitionnement spatial multicoeur. Notre adaptation du noyau n'offre aucune garantie de protection des accès utilisateurs légaux, mais possiblement invalides.	4.9.4, p.113
Les mécanismes de protection d'accès concurrents aux objets fournis par le noyau sont insuffisants pour garantir la validité des opérations sur les ressources partagées. Les développeurs de partitions doivent fournir des mécanismes de synchronisation additionnels au niveau applicatif pour faire respecter l'ordonnancement des opérations.	4.9.4, p.114

Description	Section
Les verrous internes des objets du noyau utilisent l'algorithme « spinlock », qui ne fournit pas de garanties d'accès équitable aux ressources. Les programmeurs de partitions doivent gérer l'accès aux ressources partagées de manière à garantir un accès équitable à une ressource partagée, le cas échéant.	4.9.4, p.115

CHAPITRE 5

RÉALISATION DE XTRATUM-PPC SUR POWERPC

5.1 Survol

Ce chapitre couvre l'implémentation de XtratuM-PPC sur un processeur PowerPC multicoeur. Notre cible finale est le SoC Freescale MPC8641D, doté de deux coeurs PowerPC e600.

Nous couvrons la réalisation de XtratuM-PPC en suivant l'ordre logique d'implémentation du prototype. Nous débutons avec le démarrage du système et progressons jusqu'à la mise à l'essai d'un plan d'exécution hybride sur le prototype complété.

Nous avons déjà abordé les adaptations architecturales liées au partitionnement spatial et temporel dans le chapitre précédent. Au niveau de la réalisation sur PowerPC, nous ne reviendrons pas en détail sur ces adaptations génériques. Nous présenterons plutôt les détails de l'exploitation du MMU du coeur e600, ainsi que l'implémentation du changement de contexte de partition. Ces deux aspects principaux de l'implémentation sont respectivement à la base du partitionnement spatial et du partitionnement temporel.

Finalement, nous abordons une étude de cas à la section 5.10. Cette dernière a été construite afin de valider l'implémentation finale de notre prototype à l'aide d'un système IMA synthétique exploitant plusieurs coeurs. En outre, trois hypothèses de performance du noyau sont validées à travers cette étude de cas.

5.2 Plateforme cible

La réalisation de XtratuM-PPC vise les processeurs PowerPC multicoeurs. Il existe plusieurs SoC commerciaux qui intègrent de multiples coeurs PowerPC. Dans cette catégorie, les SoC

MPC8572E et MPC8641D sont mentionnés à plusieurs reprises dans la littérature sur l'évaluation des processeurs multicoeurs en avionique et dans l'industrie avionique [21, 62, 43]. En particulier, le MPC8572E est déjà supporté par le système d'exploitation IMA PikeOS [62]. Ces deux SoC sont dotés de doubles coeurs à PowerPC 32-bit. De plus, ces pièces sont supportées par la plateforme virtuelle Simics (voir section 5.2.2 plus loin). Après l'évaluation de ces alternatives, nous avons choisi de cibler le MPC8641D de Freescale pour deux raisons :

1. la version académique de la plateforme virtuelle Simics le supporte ;
2. son architecture générale est très similaire aux autres pièces multicoeurs embarquées de Freescale (MPC8572E, P10xx, P20xx et P40xx).

Dans le reste de cette section, nous présentons un survol de notre plateforme cible, sur laquelle XtratuM-PPC est réalisé.

5.2.1 Architecture du PowerPC MPC8641D

Le PowerPC MPC8641D de Freescale est un processeur avancé pour applications générales. Il est réalisé sous la forme d'un SoC comprenant toutes les interfaces nécessaires à la réalisation d'un système complet basé sur une seule puce principale. Il comporte deux coeurs PowerPC e600 32-bit cadencés à 1.5 GHz, leurs caches, un bus partagé, deux contrôleurs de mémoire et une multitude de périphériques communs dans les systèmes embarqués. Le schéma-bloc complet du MPC8641D est présenté à la figure 5.1.

Le coeur PowerPC e600 est un processeur superscalaire de la famille PowerPC classique. Il est doté de 11 unités de traitement : 4 unités entières, 4 unités vectorielles, une unité de calcul à point flottant (« floating point unit » ou FPU, en anglais), une unité « load-store » et une unité de branchement. En plus d'être superscalaire, ce coeur comporte plusieurs caractéristiques avancées pour la maximisation des performances :

- caches L1 séparées (pseudo Harvard) de 32 ko chacune pour les instructions et les données ;

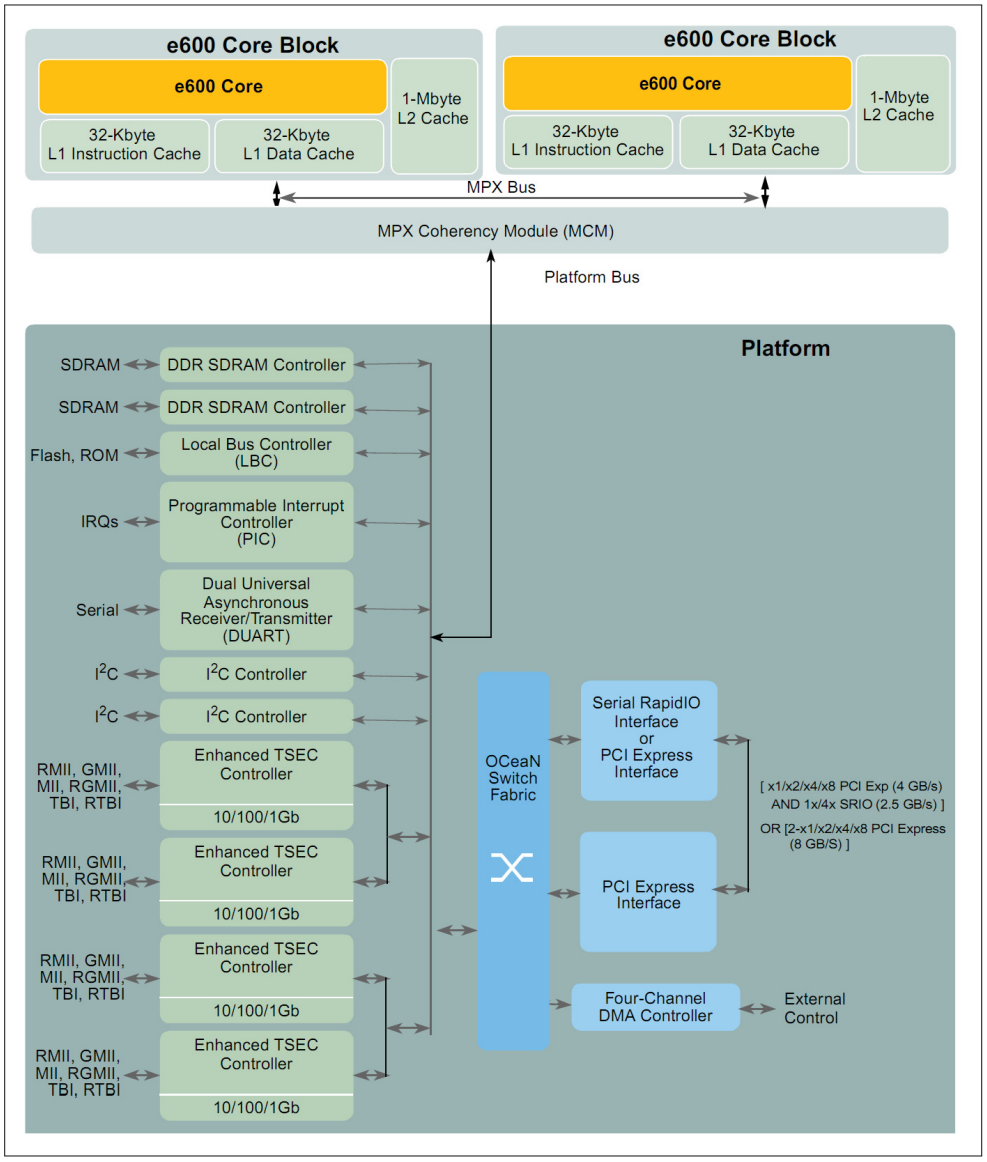


Figure 5.1 Diagramme-bloc du MPC8641D, tirée de [29, p. 1-5]
avec la permission de Freescale Semiconductors

- MMU séparés (pseudo Harvard) ;
- cache L2 unifiée de 1 Mo ;
- unité de prédiction des branchements avec exécution spéculative ;
- cohérence de cache matérielle à protocole MESI ;
- opérations atomiques en deux étapes avec protocole LL/SC [59, p.257].

Il est important de mentionner que toutes ces caractéristiques affectent la prédictibilité du délai d'exécution, mais elles sont cependant essentielles à l'atteinte du niveau de performances attendu de ce type de processeur [43, 21, 37].

Les caractéristiques des caches et des coeurs du MPC8641D ont été analysées par une équipe de recherche mixte de l'industrie avionique et automobile [21]. D'après ces analyses, le processeur MPC8641D peut être considéré comme prédictible par rapport à l'analyse statique avec des outils actuels si deux conditions essentielles sont respectées : 1) un noyau de partitionnement robuste du type de ceux employés en IMA effectue un ordonnancement statique calculé pour éviter les partages de ressources nuisibles et 2) les caches sont verrouillées lors des fenêtres d'activation des partitions. Ces conditions peuvent être respectées par notre implémentation de XtratuM-PPC.

5.2.2 Plateforme virtuelle

Nous avons utilisé une plateforme virtuelle (en anglais « Virtual Platform » ou VP) pour l'ensemble du processus d'implémentation du noyau XtratuM-PPC puisque l'évaluation de cette technologie était l'un des objectifs de nos travaux. Une plateforme virtuelle est un logiciel regroupant des modèles de simulation de processeurs (« Instruction Set Simulators » ou ISS), de bus de communication, de périphériques et de mémoires. Les couches conceptuelles d'une VP sont présentées à la figure 5.2. Cet ensemble de modèles est intégré afin de simuler une plateforme matérielle particulière avec un niveau élevé de fidélité sur un ordinateur de bureau.

Puisque l'utilisation de plateformes virtuelles est un sujet assez lourd et que les conséquences de leur utilisation sont indépendantes de nos travaux d'adaptation multicoeur de XtratuM, nous limitons notre couverture dans ce mémoire au strict minimum. Notre objectif de recherche visant l'évaluation des plateformes virtuelles pour le développement d'applications embarquées a fait l'objet de travaux connexes, qui ont été publiés et présentés à la conférence « 2011 IEEE

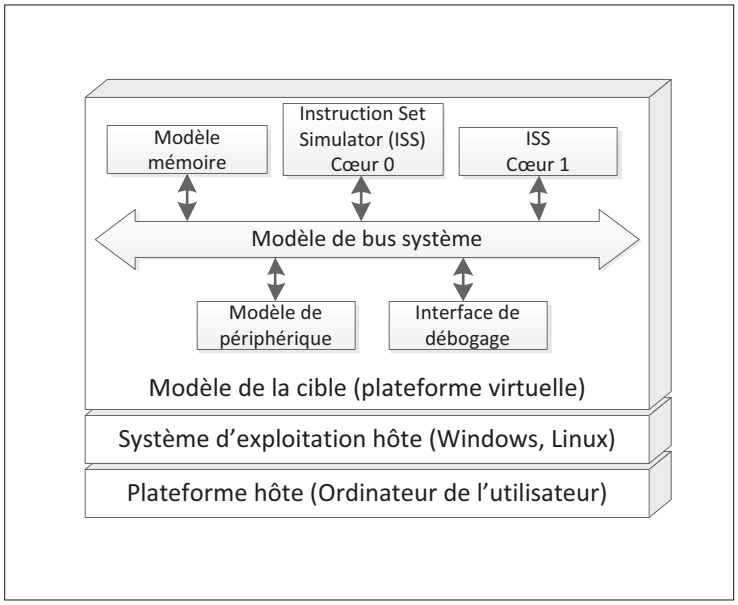


Figure 5.2 Couches conceptuelles d'une plateforme virtuelle

International Symposium on Rapid System Prototyping » [18]. Notre article est reproduit à l'annexe II à la page 209.

5.2.2.1 Motivations

L'utilisation d'une VP était motivée dès le départ pour une raison pragmatique : plusieurs VP sont disponibles gratuitement dans les environnements académiques, ce qui réduit sensiblement les coûts en comparaison avec l'achat d'une plateforme matérielle. En plus d'être gratuites, toutes les plateformes virtuelles que nous avons évaluées offraient des outils de débogage avancés. En particulier, les VP permettent d'observer de manière non intrusive l'état entier du système. Cette caractéristique est particulièrement utile lorsqu'on développe des logiciels embarqués de bas niveau, tels que des noyaux, où la majorité des bogues se retrouvent dans des modules qui affectent l'état global du système et pour lesquels aucune visibilité intrinsèque ne peut être fournie.

En plus des considérations pragmatiques, nous avons choisi de travailler avec une VP afin de construire une étude de cas qualitative de leur utilisation pour le développement embar-

qué en IMA. Les VP offrent plusieurs opportunités pour l'amélioration de la productivité du développement embarqué en comparaison avec l'approche traditionnelle de l'utilisation de sondes matérielles, tel que les sondes JTAG. Les plus importantes opportunités sont les suivantes [15, 25, 63] :

- elles permettent d'automatiser les tests logiciels ;
- elles permettent d'instrumenter les coeurs et les périphériques afin d'obtenir de l'information précise sur les évènements découlant de l'exécution d'un logiciel ;
- elles permettent la modification arbitraire de tous les registres de la machine, incluant ceux impossibles à accéder en direct par des sondes matérielles.

5.2.2.2 Choix de la plateforme virtuelle

Avant le début des travaux, nous avons évalué plusieurs plateformes virtuelles afin de déterminer laquelle serait la plus appropriée à nos travaux. En particulier, nous avons évalué MC-ISS (« Multi-Core Instruction Set Simulator ») d'IBM, Simics de Wind River et le projet à code source libre QEMU. Le produit Platform Architect de Synopsis, très populaire en industrie, n'avait pas été évalué, car il n'était pas immédiatement disponible en version académique.

Très rapidement, Simics s'est démarquée du lot, car elle était la seule à offrir les caractéristiques suivantes :

- support des processeurs PowerPC multicoeurs MPC8641D, MPC8572E et P4080 de Freescale, incluant tous les modules de périphériques ;
- support de l'exécution renversée ;
- disponibilité de forums de support en ligne ;
- possibilité d'exécuter des scripts directement dans l'environnement de simulation.

Ces quatre caractéristiques, dont la plus importante était le support des processeurs MPC8641D et MPC8572E, n'étaient disponibles qu'avec Simics. De plus, les autres VP évaluées auraient

nécessité le développement de plusieurs modèles de simulation additionnels pour supporter nos objectifs.

5.2.3 Configuration de la plateforme cible sous Simics

Nous avons configuré la plateforme virtuelle cible afin de supporter nos besoins de développement.

En premier lieu, nous avons sélectionné un modèle de simulation fonctionnelle réduit sans caches. Le choix d'éliminer les caches était nécessaire pour des raisons de vitesse d'exécution : la simulation fonctionnelle avec un modèle réel de caches est ralentie d'un facteur supérieur à 10. Puisque nous n'effectuons aucune mesure de performances et qu'il est déjà établi que les caches causent des problèmes de détermination du temps d'exécution, nous n'en avions pas besoin pour réaliser le prototype logiciel de l'adaptation. Il demeure possible d'ajouter les caches au modèle dans une étape ultérieure des travaux. La figure 5.3 présente un diagramme-bloc du sous-ensemble du MPC8641D modélisé pour nos travaux. Notons que notre modèle de simulation est doté de 3 coeurs alors que la puce réelle n'en contient que deux. Ce choix est permis par la plateforme virtuelle, qui supporte jusqu'à 8 coeurs. L'utilisation de trois coeurs a permis de tester la généralité de notre réalisation.

En deuxième lieu, nous avons produit des scripts de configuration qui fournissent les outils de débogage nécessaires. En particulier, nous avons configuré les outils suivants :

- une interface au logiciel de débogage GDB sur chaque coeur ;
- les tables de symboles liées aux binaires de tests ;
- les points d'instrumentation principaux ;
- le démarrage automatisé du noyau.

Après ces configurations, l'exécution d'une version de test du noyau XtratuM-PPC s'effectuait en un seul clic.

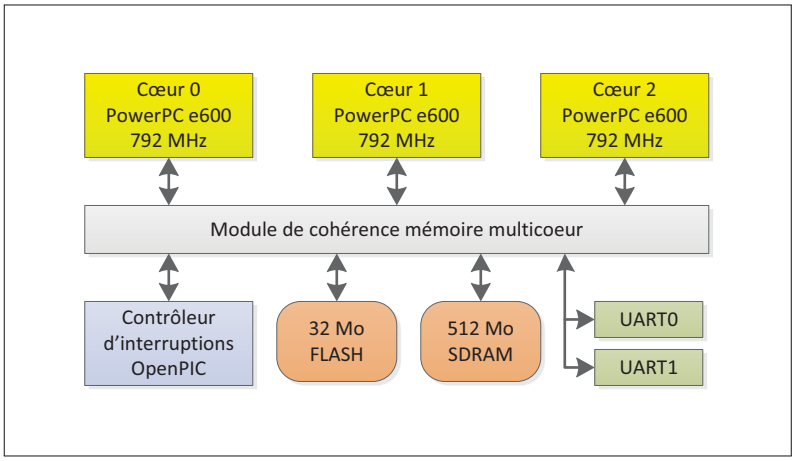


Figure 5.3 Diagramme-bloc du modèle de simulation utilisé

5.2.4 Chaîne de compilation

Le noyau XtratuM-PPC est écrit en langage C et en assembleur PowerPC. Nous avons utilisé la chaîne de compilation Sourcery-G++ pour PowerPC pour l'ensemble des travaux. Cette chaîne de compilation est basée sur les outils de la suite GNU GCC, qui sont les plus répandus en développement embarqué sur PowerPC. L'interface binaire d'application choisie est PowerPC-EABI, soit celle dédiée aux applications embarquées. Pour l'ensemble du développement, nous avons utilisé le mode d'optimisation `-O2` (« optimize more ») et la génération de symboles de débogage `-g3` de GCC. Le tableau 5.1 résume les versions des outils utilisés pour la réalisation de XtratuM-PPC.

Tableau 5.1 Outils utilisés pour la réalisation XtratuM-PPC

Fonction	Outil	Version
Compilateur C croisé PowerPC	GNU powerpc-eabi-gcc	4.4.1
Éditeur de lien croisé PowerPC	GNU powerpc-eabi-ld	2.19.51
Assembleur croisé PowerPC	GNU powerpc-eabi-as	2.19.51
Débogueur croisé PowerPC	GNU powerpc-eabi-gdb	6.8.50
Compilateur C	GNU gcc	3.4.4
Make	GNU Make	3.81
Simulateur fonctionnel	Wind River Simics	4.0.60
Modèle PowerPC e600	Wind River Simics	4.0.12

5.3 Démarrage du système

Le démarrage du système comprend toutes les étapes jusqu'au relâchement de l'ordonnanceur de partitions sur tous les coeurs. Nous avons déjà couvert les aspects génériques du démarrage à la section 4.6.2. Dans cette section, nous présentons les spécificités du démarrage sur un processeur MPC8641D. Le démarrage de XtratuM-PPC est la partie du noyau ayant nécessité le plus grand nombre d'adaptations par rapport à la version originale LEON. Il s'agit d'une procédure complexe qui requiert plusieurs étapes de configurations de bas niveau. Les prochaines sections décrivent en détail la réalisation du démarrage multicoeur du noyau XtratuM-PPC.

5.3.1 Vecteurs d'exceptions

Les coeurs PowerPC e600 possèdent une table de vecteurs de 32 entrées de 256 octets chacune. Puisque toutes les instructions des PowerPC font 4 octets, chaque vecteur peut comporter un maximum de 64 instructions. Le tableau 5.2 présente la liste des huit premiers vecteurs d'exceptions. Chaque entrée se trouve à une adresse de décalage fixe en fonction du type d'exception. L'adresse de base de la table de vecteurs est soit en mémoire haute à l'adresse 0xFFF0_0000, soit en mémoire basse à l'adresse 0x0000_0000. Un bit du registre de contrôle de la machine permet aux systèmes d'exploitation de configurer l'adresse de base utilisée. Le cas par défaut place la table en mémoire haute. Les processeurs PowerPC sont préconfigurés afin de pouvoir minimalement accéder à une ROM externe à l'adresse 0xFFF0_0000 dès le démarrage.

Lors du démarrage à froid de la machine, un saut implicite au vecteur de démarrage à l'adresse 0xFFF0_0100 est effectué. La ROM sur le bus externe du processeur doit contenir des instructions valides à cette adresse. Ces instructions seront les premières exécutées et serviront à initialiser la machine. Habituellement, le vecteur de démarrage configure la cache L1 de données en mode de mémoire RAM tampon en verrouillant une voie, avant que les contrôleurs

Tableau 5.2 Liste des huit premiers vecteurs d'exceptions du PowerPC

Adresse de décalage	Exception
0x0000	Réservée
0x0100	Démarrage système
0x0200	Erreur critique (« Machine check »)
0x0300	Accès invalide de donnée
0x0400	Accès invalide d'instruction
0x0500	Interruption externe
0x0600	Alignement invalide
0x0700	Exception de programme

de mémoire ne soient configurés, pour ensuite sauter plus loin en ROM vers une procédure de démarrage complète.

Afin de supporter différents cas d'utilisation multicoeur, le processeur MPC8641D comporte deux modes de réadressage des vecteurs d'exceptions. Ces modes de réadressage offrent une flexibilité additionnelle et permettent surtout d'utiliser des vecteurs d'exceptions n'importe où dans la carte mémoire.

Le premier mode de réadressage permet de décaler les accès physiques au premier segment de 256 Mo de la mémoire basse (0x0000_0000–0x0FFF_FFFF) vers le segment suivant (0x1000_0000–0x1FFF_FFFF) pour le deuxième coeur. Avec ce mécanisme, il devient possible pour chacun des deux coeurs d'exploiter leurs propres vecteurs d'exceptions de mémoire basse, ainsi qu'un espace suffisant pour les données internes d'un système d'exploitation.

Le deuxième mode de réadressage est plus subtil : il ne permet que de décaler les accès physiques à la page de 4k des exceptions en mémoire haute (0xFFF0_0000–0xFFF0_0FFF). Ce mode appelé « Boot Page Translation » permet de rediriger cette page vers n'importe quelle région de l'espace d'adresses physiques avec une granularité d'alignement de 1 Mo. Il permet ainsi de déplacer uniquement les vecteurs d'exceptions critiques (démarrage, interruptions, etc.) pour tous les coeurs. À l'exception du démarrage et de la gestion des exceptions, tous les autres accès demeurent inchangés. Il est donc possible pour un système d'exploitation de

copier une version modifiée des vecteurs dans la mémoire et de s'en servir pour le démarrage des autres coeurs.

Le placement des vecteurs d'exceptions pour XtratuM-PPC emploie le mode de réadressage « Boot Page Translation » pour le démarrage des coeurs après le coeur 0. De plus, lors du démarrage, les coeurs sont configurés pour dorénavant utiliser les vecteurs en mémoire basse. Le processus complet de démarrage est présenté dans la prochaine section.

5.3.2 Procédure de démarrage

Le démarrage de XtratuM-PPC est réalisé à l'aide de deux chargeurs utilisés en cascade. Nous avons déjà mentionné le chargeur RSW de XtratuM à plusieurs reprises. Ce chargeur est directement écrit en ROM et démarre immédiatement dans le cas des processeurs LEON. Cependant, les processeurs MPC8641D sont beaucoup plus complexes que les LEON. Pour cette raison, le code d'initialisation de RSW ne suffit pas à configurer la plateforme dans un état suffisamment avancé pour permettre à XtratuM d'être copié en mémoire.

Les processeurs complexes sont souvent accompagnés d'un micrologiciel de démarrage système (« bootloader firmware », en anglais) exécuté en ROM ou Flash au démarrage du processeur. Ce bootloader sert à configurer les interfaces mémoires et l'ensemble minimum de périphériques et fonctions du processeur nécessaires au chargement ultérieur des noyaux de systèmes d'exploitation. Nous avons utilisé le bootloader U-Boot pour préparer la machine à exécuter RSW. U-Boot est directement supporté et développé par Freescale pour leurs processeurs embarqués. La version 1.3.0 de U-Boot que nous avons utilisée supporte les processeurs MPC8641D. Pour démarrer le noyau, nous envoyons des commandes à U-Boot afin qu'il exécute le chargeur RSW. Ce dernier est désormais capable de charger XtratuM en mémoire puisque les interfaces RAM ont été configurées par U-Boot.

Le démarrage multicoeur est réalisé de deux manières différentes, selon qu'il s'agisse du coeur principal ou des autres coeurs. Le coeur principal doit préparer le démarrage des coeurs ad-

ditionnels, car son vecteur de démarrage par défaut, en mémoire haute, force l'exécution de U-Boot en ROM. U-boot n'implémente aucune des fonctions nécessaires au démarrage multicoeur ordonné de XtratuM-PPC.

Le processeur principal suit les étapes suivantes pour son démarrage :

1. Le processeur démarre et exécute l'instruction à l'adresse physique 0xFFF0_0100. Dans notre cas, cette adresse arrive dans une ROM qui contient U-Boot.
2. U-Boot configure la machine.
3. U-Boot charge RSW en mémoire à partir de la ROM.
4. U-Boot démarre RSW.
5. RSW charge en mémoire le noyau XtratuM-PPC, la table de configuration binaire et les partitions.
6. RSW saute vers la procédure de démarrage principale de XtratuM-PPC (`StartXM`).
7. La procédure de démarrage de haut niveau de XtratuM-PPC (`Setup`) arrive au point où les autres coeurs sont relâchés.

Lors du relâchement des autres coeurs, la procédure suivante intervient :

1. Le « Boot Page Translation » est configuré par le coeur 0 afin de pointer vers la mémoire basse (0x0000_0000).
2. Les vecteurs d'exceptions de XtratuM-PPC sont copiés vers 0x0000_0000–0x0000_1FFF en mémoire basse.
3. Les autres coeurs sont activés par le coeur 0 et exécutent chacun leur vecteur de démarrage à l'adresse 0x0000_0100. En raison du réadressage, ils croient accéder au vecteur par défaut à 0xFFF0_0100.
4. Le vecteur de démarrage saute vers la procédure de démarrage principale de XtratuM-PPC (`StartXM`), comme l'étape 6 du démarrage du coeur principal.

À partir de ce moment, tous les coeurs sont activés. La procédure de démarrage de haut niveau présentée à la section 4.6.2 termine ensuite le démarrage générique.

Le point d'entrée du noyau est la procédure `StartXM`, exécutée à l'étape 6 du démarrage du coeur 0 et à l'étape 4 du démarrage des autres coeurs. Cette procédure vise à préparer l'environnement logiciel pour l'exécution du code C compilé du noyau. Les étapes de préparation sont les suivantes :

1. Configuration de la carte mémoire initiale du noyau dans le MMU.
2. Invalidation du TLB.
3. Configuration et vidange des caches L1 et L2.
4. Mise en route de l'adressage virtuel.
5. Réinitialisation des autres coeurs si le coeur actuel est le coeur 0.
6. Configuration des registres systèmes du processeur.
7. Préparation des données du binaire du noyau.
8. Configuration du descripteur de fil d'exécution pour le noyau actuel (duplication du fil d'exécution du noyau), incluant l'identification du coeur.
9. Configuration de la pile.
10. Saut au début du code C compilé du noyau.

Après ces préparations, le code C du noyau est exécuté en sautant à la fonction `Setup`. La figure 5.4 présente un résumé du cheminement du démarrage de XtratuM-PPC sur tous les coeurs, incluant le chargement par U-Boot. On y remarque qu'éventuellement tous les coeurs finissent au même point, soit à l'exécution de `StartXM` et ensuite `Setup`.

5.3.3 Identification des coeurs

Les étapes 5 et 8 de la procédure `StartXM` dépendent de l'identification du coeur. En effet, le démarrage principal est effectué à l'aide du coeur 0 avant le démarrage des autres coeurs.

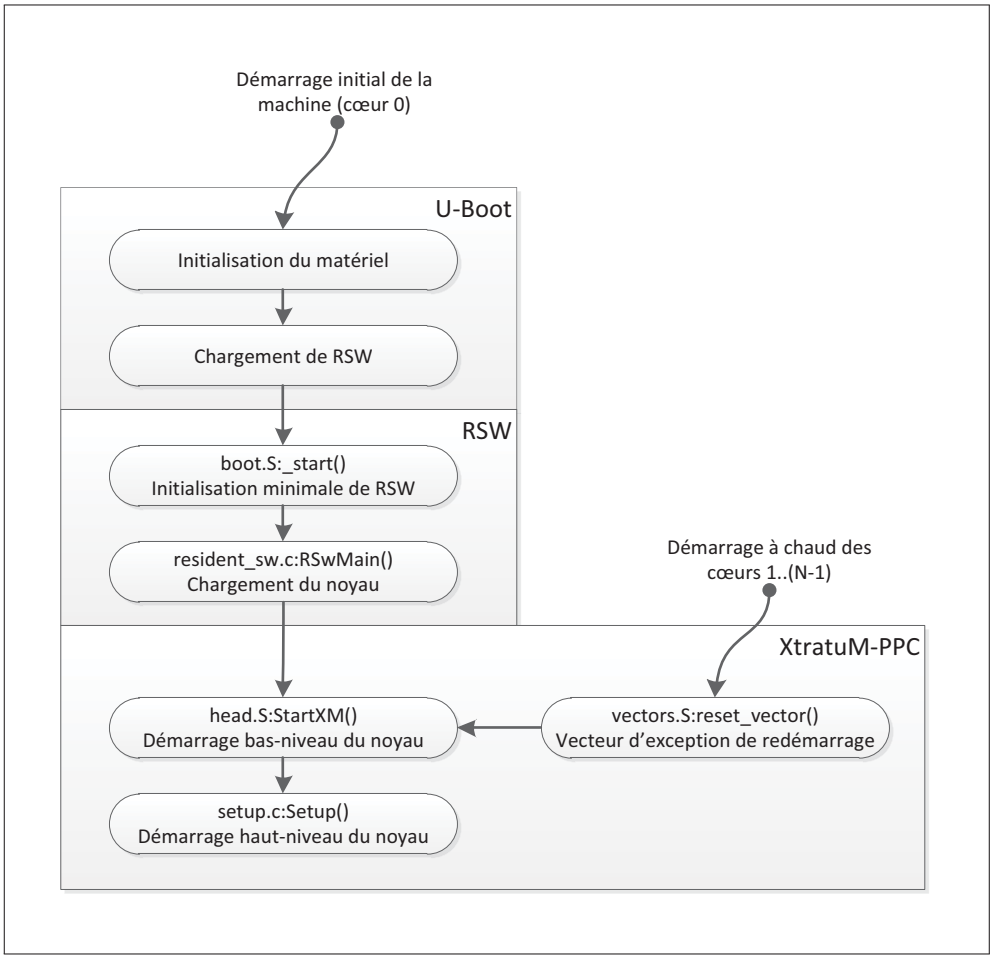


Figure 5.4 Cheminement du démarrage du noyau XtratuM sur tous les coeurs

De plus, les structures de données locales en « thread-local storage » nécessaires au support de la duplication du fil d'exécution du noyau sur chaque coeur sont indexées en fonction de cet identifiant.

L'identification des coeurs n'est pas figée dans ces derniers, puisqu'ils sont tous exactement identiques. Un mécanisme extérieur aux coeurs doit être utilisé pour obtenir leur identification. La méthode que nous avons développée à cet effet est similaire à celle utilisée par le noyau Linux.

Les coeurs e600 possèdent un registre interne nommé `PIR` (« Processor Identification Register »). Ce registre peut contenir une valeur arbitraire de 32 bits et n'est accessible que par

des instructions d'accès aux registres spéciaux du processeur [1]. Nous utilisons ce registre pour enregistrer dans un endroit globalement et facilement accessible l'identifiant séquentiel des coeurs. Le registre spécial `PIR` peut ensuite être copié dans un registre général du processeur pour réaliser l'indexage propre à chaque coeur, sans devoir effectuer d'accès coûteux à la mémoire.

L'identifiant en tant que tel est obtenu par la lecture du registre `WHOAMI` dans le contrôleur d'interruptions OpenPIC. Ce registre est câblé dans un espace d'adressage commun dont l'accès est guidé en fonction du coeur qui effectue l'accès. Autrement dit, chaque processeur utilise la même adresse pour l'accès, mais chacun lit une version qui lui est propre en fonction de sa position de câblage sur le bus interne du SoC. Les adresses propres à chaque coeur sont autogénérées par le matériel. La valeur lue dans `WHOAMI` est copiée dans `PIR` et utilisable globalement à partir de ce moment.

5.3.4 Problèmes potentiels

Nous avons identifié des problèmes de sûreté potentiels dans la procédure de démarrage multicoeur.

Le premier problème est causé par le démarrage à travers un vecteur d'exception déplacé en RAM. En effet, seul le coeur 0 passe dans U-Boot en ROM et il ne le fait seulement que lors de son démarrage initial. Une faute pourrait survenir lors de l'exécution de code dans la page des vecteurs d'exceptions en raison d'un dépassement de capacité de tampon ou d'une corruption de cellule mémoire. Le résultat serait assurément une demande de redémarrage par le moniteur de santé du système, à moins que le code de ce dernier soit aussi corrompu. Puisque la page d'exceptions en RAM est présumée corrompue, on ne peut plus s'y fier. On ne peut donc pas garantir un redémarrage adéquat du coeur en question. Ce problème serait

1. Les instructions `mtspr` (move to special-purpose register) et `mfspr` (move from special-purpose register) sont utilisées à cet effet sur les PowerPC.

beaucoup moins probable si les vecteurs d'exceptions se trouvaient en ROM. Nous avons choisi d'utiliser la RAM pour les vecteurs, car nous ne voulions pas modifier U-Boot pour supporter tous les détails de notre démarrage multicoeur. Nous avons préféré découpler ces détails du bootloader. U-Boot est déjà programmé pour supporter le démarrage multicoeur de Linux. Le problème pourrait être réglé en produisant un bootloader minimal supportant la procédure de démarrage multicoeur de XtratuM-PPC et le strict minimum de configuration système nécessaire au chargement de RSW.

Une catégorie additionnelle de problèmes est due à l'architecture du MPC8641. Les registres contrôlant le réadressage ainsi que le démarrage des coeurs sont globalement accessibles par les coeurs. Il serait possible qu'une défaillance cause une écriture intempestive dans un de ces registres. Cela pourrait engendrer toutes sortes d'effets, notamment le redémarrage inopiné d'un ou plusieurs coeurs. Aucun verrouillage de ces registres critiques n'est possible et aucun protocole de protection d'écriture par mot de passe n'est disponible. Ce problème est corroboré par les résultats de Mahapatra *et al.* [44] obtenus par l'évaluation du processeur MPC8572 qui comporte les mêmes mécanismes de réadressage et de démarrage multicoeur.

5.4 Gestion du temps

Les pilotes de gestion du temps sont utilisés par tous les mécanismes de partitionnement temporel dans XtratruM-PPC. L'architecture originale de XtratuM était assez flexible pour permettre la réalisation transparente d'une version compatible avec le PowerPC. Le processeur MPC8641 est doté d'horloges et de minuteries plus flexibles que celle des processeurs LEON, ce qui a facilité l'adaptation.

5.4.1 Horloges locales

Pour les horloges locales, nous avons utilisé le mécanisme architectural de base de temps (« Timebase Counter » ou TBC) présent sur tous les processeurs PowerPC. Chaque coeur est doté

d'un TBC à haute résolution de 64 bits. Ce dernier peut directement être lu et écrit à l'aide des instructions d'accès aux registres spéciaux. La source d'horloge du TBC est l'horloge de bus du processeur, divisée par quatre. Dans le cas d'un MPC8641 cadencé à 792 MHz (l'horloge de notre plateforme cible), l'horloge de bus est le tiers de cette valeur, soit 264 MHz, donc la source du TBC est de 66 MHz. Cette source permet une résolution temporelle de 15.15 ns pour l'horloge locale, soit beaucoup mieux que la résolution prescrite pour les horloges dans XtratuM (1 μs). Le pilote de l'horloge locale de chaque coeur garde en mémoire un facteur de conversion entre la valeur du TBC et les unités temporelles en microsecondes. Le facteur de conversion est basé sur des informations de vitesse d'horloge fournies par U-Boot lors du démarrage.

5.4.2 Minuteries

L'instance locale de chaque noyau emploie une minuterie locale. Nous avons utilisé le décompteur architectural (« decrementer ») du PowerPC pour réaliser cette minuterie locale. Ce décompteur est d'une largeur de 32 bits et il génère une exception masquable chaque fois que le compte passe de 0x0000_0000 à 0xFFFF_FFFF. Le décompteur utilise la même source d'horloge que le TBC. Le gestionnaire de l'interruption du décompteur appelle le même code générique de gestion des minuteries que celui utilisé pour les processeurs LEON.

En plus du décompteur de chaque coeur, le processeur MPC8641 est doté de quatre minuteries globales de 32 bits dans le contrôleur d'interruptions OpenPIC. Ces minuteries peuvent interrompre toutes les combinaisons de coeurs au même moment. Nous avons réalisé un pilote pour ces minuteries globales, mais elles ne sont pas actuellement utilisées.

5.4.3 Synchronisation des horloges locales

Le MPC8641 ne comporte aucune horloge globale à haute résolution. Les horloges locales doivent donc être synchronisées très précisément, sans quoi il risque d'y avoir une très longue

variation entre les moments d'exécution des ordonnanceurs sur les différents coeurs. Plus la différence est grande, plus la gigue temporelle entre les coeurs est grande. À la limite, cette gigue pourrait empêcher le respect d'un plan d'exécution conçu pour éliminer les partages de ressources interdits entre les coeurs.

Nous avons réalisé un algorithme de synchronisation des horloges locales pour limiter la gigue temporelle entre les coeurs à moins d'une microseconde entre les deux coeurs les plus éloignés. L'algorithme consiste en l'enregistrement de plusieurs valeurs d'horloge locale dans un tampon en cache sur chaque processeur à l'aide d'une boucle serrée. Par la suite, les coeurs se rencontrent à une barrière et le coeur 0 calcule un décalage moyen entre son horloge locale et celle de chaque coeur. Ce décalage est ajouté par chaque coeur à son horloge locale, ce qui synchronise les horloges avec une grande précision. Notre algorithme est fonctionnel sur la plateforme virtuelle, mais n'a pas été testé sur le matériel. Cet algorithme est très dépendant de la hiérarchie mémoire et des délais d'accès, donc il risque d'être moins performant sur la vraie plateforme. Il est cependant possible de borner son temps d'exécution.

Il serait possible d'adapter l'algorithme de synchronisation des bases de temps utilisé dans Linux. Cet algorithme est très précis et fiable. De plus, il est largement déployé et éprouvé. D'après nos communications privées avec Samuel Rydh d'Intel, le créateur de cet algorithme pour l'architecture PowerPC, cet algorithme serait fiable sur tous les processeurs PowerPC multicoeurs dont le protocole de cohérence des caches n'est pas priorisé en faveur d'un coeur particulier. Cependant, il s'agit d'un algorithme non déterministe. Sa durée est incalculable à moins de forcer une sortie de boucle anticipée, qui élimine les garanties quant à la précision de l'algorithme.

Un autre type d'algorithme, basé sur la lecture compétitive d'un compteur global, tel qu'un des compteurs de minuteries globales du module OpenPIC ou du compteur de performance

générique du MPC8641, pourrait aussi être évalué. Ce genre d'algorithme devrait être essayé sur le matériel, auquel nous n'avions pas accès.

Des tests additionnels sur le matériel devraient être réalisés afin de déterminer quelle méthode serait la meilleure pour la synchronisation des horloges.

5.4.4 Problèmes potentiels

En plus des problèmes architecturaux du noyau liés à la gestion du temps présentés aux sections 4.8.4.2 et 4.8.4.3, nous avons identifié deux problèmes causés par notre stratégie d'implémentation de la gestion du temps sur le MPC8641.

Puisque les coeurs ne peuvent pas accéder directement et avec une faible latence à une horloge globale fiable, nous devons employer l'horloge locale de chaque coeur pour la gestion du temps et l'exécution des ordonnanceurs locaux. Malheureusement, la synchronisation assez précise de ces horloges locales peut être très longue, comme nous l'avons déjà mentionné à la section précédente. De plus, les algorithmes de synchronisation évalués requièrent tous une exécution simultanée sur chaque coeur, c'est-à-dire qu'il n'est pas possible de synchroniser les horloges sans le rendez-vous temporel d'au moins les deux coeurs impliqués.

Le premier problème identifié est la subtilisation de temps causée par le besoin de resynchroniser les horloges lors du redémarrage d'un coeur causé par exemple en raison d'une défaillance. Peu importe l'algorithme de synchronisation employé, il reste que les horloges locales doivent être resynchronisées éventuellement. Même avec un algorithme de synchronisation très rapide, il serait tout de même nécessaire d'interrompre un coeur en traitement pour la durée de la synchronisation. La subtilisation de temps de partition résultant de cette synchronisation pourrait être évitée en gardant des tranches de temps assignées uniquement à la resynchronisation des coeurs dans le plan d'exécution. Cependant, le support de cette solution requerrait des changements au noyau afin de supporter ces tranches de temps spéciales. De plus, le coeur redémarré

pourrait attendre un long délai avant l'arrivée de la prochaine fenêtre de resynchronisation, ce qui rendrait l'analyse de la gestion des défaillances plus complexe. Une alternative qui n'a pas été mise à l'essai est l'ajout d'une phase de resynchronisation d'horloge à la frontière de chaque bloc d'activation majeur.

Le deuxième problème identifié est la perte des garanties d'isolation temporelle entre les coeurs en cas de désynchronisation des horloges locales. La stabilité de la synchronisation des horloges locales, établie au démarrage du noyau, est strictement nécessaire pour garantir l'intégrité du partitionnement temporel et le respect du plan d'exécution. Si la moindre désynchronisation des horloges locales survenait, le partitionnement temporel multicoeur avec notre modèle de noyau SMP serait immédiatement compromis. Cela est dû à la supposition dans notre adaptation multicoeur que le plan d'exécution est synchronisé entre les coeurs, ce qui permet de garantir, par exemple, des fenêtres d'activation exclusives où un seul coeur exécute une partition critique sans interférence.

Comme le démontrent ces deux problèmes, le compromis technique de l'utilisation des horloges locales sur chaque coeur n'est pas sans conséquence. L'absence d'une horloge globale accessible rapidement par tous les coeurs est le maillon faible de la gestion du temps dans la cible MPC8641. L'utilisation d'une horloge globale à haute résolution rapidement accessible et également prioritaire entre les coeurs pourrait remédier à ces problèmes. Un autre solution matérielle potentielle serait l'implémentation au sein des coeurs d'une horloge locale dotée d'une fonction de réinitialisation globale par signal de synchronisation externe. De même, des stratégies plus avancées de synchronisation périodique pourraient être évaluées pour mitiger certaines des conséquences de l'utilisation d'horloges locales.

5.5 Gestion des interruptions

Nous avons développé une gestion des interruptions basée sur le contrôleur d'interruptions OpenPIC. Le contrôleur d'interruptions sert à gérer la répartition des interruptions à des sous-

gestionnaires sur le coeur approprié et à faire respecter un ensemble de priorités entre les sources.

Les mécanismes de gestion des interruptions sur les processeurs PowerPC diffèrent de ceux des processeurs LEON. En particulier, le nombre de sources d'interruptions matérielles est beaucoup plus élevé sur le MPC8641 (84 sources) que sur un LEON2 AT697E (16 sources). De plus, les PowerPC ne possèdent pas de vecteurs d'interruptions individuels pour chaque source d'interruption externe, contrairement aux LEON. La gestion des interruptions est plutôt centralisée vers un seul gestionnaire pour toutes les interruptions externes.

Dans les sous-sections qui suivent, nous présentons les caractéristiques de notre pilote du contrôleur d'interruptions OpenPIC, la construction des gestionnaires d'interruptions et d'exceptions du noyau, ainsi que le mécanisme de traitement des interruptions et exceptions virtuelles.

5.5.1 Pilote du contrôleur d'interruptions OpenPIC

Le contrôleur d'interruptions compatible OpenPIC permet de gérer les 84 sources d'interruptions du MPC8641D. En plus de gérer les sources de périphériques internes et externes, le contrôleur comporte aussi des ressources globales qui facilitent la gestion d'un système multicoeur. À cet effet, les modules suivants sont intégrés au contrôleur :

- deux groupes de quatre minuteries globales, tels que mentionnés à la section 5.4.2 ;
- quatre canaux d'interruption de notification inter-processeurs ;
- quatre canaux d'interruption de messagerie inter-processeurs ;
- huit registres partagés d'interruption à messages signalés ;
- un système de redémarrage à chaud ou à froid de n'importe quel coeur.

Toutes les sources d'interruptions provenant des modules internes du contrôleur peuvent être destinées à plusieurs coeurs en même temps si nécessaire, c'est-à-dire que les modes multicast

et broadcast sont supportés. Lorsqu'une source d'interruption est configurée pour interrompre plusieurs coeurs en même temps, chaque coeur reçoit une notification individuelle. Chaque coeur doit accuser réception de l'interruption séparément.

Avec un contrôleur OpenPIC, chaque source d'interruption se voit assigner deux registres : un registre de vecteur et de priorité (IVPR) et un registre de destination (IDR). Le format du registre de priorité est présenté à la figure 5.5. Pour chaque source, on peut configurer le masquage (bit « MSK »), la polarité et la sensibilité au niveau ou au front (bits « P » et « S »), 16 niveaux de priorité (champ « PRIORITY ») et une valeur arbitraire de vecteur (« VECTOR »). Dans notre implémentation, le champ « VECTOR » est configuré avec un identifiant séquentiel qui permet de déterminer la source d'interruption et le vecteur à utiliser à l'aide de tables. Le registre de destination est un champ de bits où chaque bit est associé à un processeur cible qui doit être interrompu lorsque la source d'interruption est activée par un évènement.

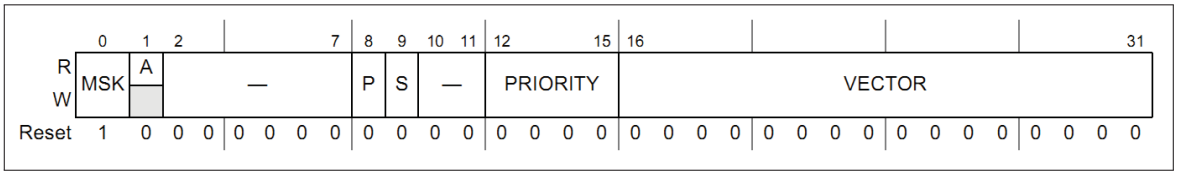


Figure 5.5 Format du registre de destination et de priorité d'une interruption dans le contrôleur OpenPIC, tirée de [29, p. 9-42] avec la permission de Freescale Semiconductors

Les registres de destination permettent avec leur format de cibler jusqu'à 28 coeurs. Les SoC les plus récents à 4 et 8 coeurs de la série QorIQ P40xx de Freescale utilisent le même contrôleur d'interruptions et exploitent les bits réservés dans le manuel du MPC8641D pour supporter les coeurs additionnels avec une interface de programmation compatible. La VP Simics supporte justement jusqu'à 8 coeurs, même sur le MPC8641D, en exploitant la symétrie apparente entre les pièces au niveau des bits réservés. Notre implémentation emploie les données fournies par les registres d'information de la plateforme pour supporter jusqu'à 8 coeurs, avec détection automatique du nombre de coeurs.

Notre pilote OpenPIC permet la gestion des minuteries globales et la configuration de toutes les sources d'interruptions. Nous avons implémenté le pilote afin de satisfaire aux interfaces logicielles abstraites de XtratuM pour la gestion des minuteries et des interruptions. Nous avons aussi développé un ensemble de tables de références qui permettent de simplifier la gestion des registres du contrôleur d'interruptions dans le noyau, afin d'éviter les cas spéciaux. Ces tables sont nécessaires, car la carte mémoire des registres du contrôleur n'assigne pas des plages d'adresses contiguës aux différentes sources d'interruptions, malgré que le format soit identique pour tous les registres. De plus, nous exploitons l'adressage automatique par coeur qui permet d'utiliser un pilote commun pour tous les coeurs. Cet adressage automatique est géré par le contrôleur OpenPIC et permet d'utiliser une adresse unique qui est automatiquement redirigée au registre assigné au coeur effectuant la transaction.

Dans la section suivante, nous présentons le traitement des interruptions dans le gestionnaire d'interruption centralisé et l'exploitation du contrôleur OpenPIC pour la vectorisation automatique des sources.

5.5.2 Gestionnaires d'interruptions et d'exceptions

L'architecture PowerPC supporte un modèle d'exceptions normalisé. Chaque exception du processeur, qu'elle soit synchrone ou asynchrone, se voit assignée à un vecteur de 256 octets dans une table de vecteurs. Pour simplifier le discours, nous différencions la dénomination des types d'exceptions du processeur en fonction de leur traitement. Nous utilisons le terme « interruption » pour les exceptions asynchrones causées par le contrôleur d'interruptions et par la minuterie locale d'un coeur. Pour toutes les autres exceptions du processeur, sauf l'appel système présenté plus loin à la section 5.7, nous utilisons le terme « exception ».

Dans notre cas, les exceptions sont toujours causées par des erreurs de programmation ou des défaillances logicielles et doivent être traitées par le moniteur de santé du système. Par exemple, le MMU du processeur génère des exceptions considérées comme normales dans les

systèmes d'exploitation traditionnels, mais notre implémentation de la protection mémoire, présentée à la section 5.8, est configurée pour que toutes les exceptions soient indicatives d'une erreur. Les interruptions doivent quant à elles être traitées par des pilotes du noyau ou dans les partitions, en fonction de la source.

Comme nous l'avons déjà mentionné, les processeurs PowerPC ne possèdent qu'un seul vecteur d'exception dédié aux interruptions externes, au lieu d'être doté d'un vecteur par source comme dans le cas des processeurs LEON. Lorsqu'une interruption autre que celle de la minuterie locale survient, le contrôleur d'interruptions doit être interrogé afin d'en déterminer la cause. Un gestionnaire spécifique peut ensuite être appelé si l'interruption avait été configurée dans le système. Autrement, un gestionnaire par défaut est appelé afin d'avertir le moniteur de santé du système qu'une interruption imprévue est survenue.

Les vecteurs d'exceptions et d'interruptions, sauf celui de l'appel système employé pour les hyperappels, sont tous basés sur un modèle similaire. Chaque vecteur doit sauvegarder le contexte du processeur, enregistrer la cause d'exception ou d'interruption et appeler la fonction gestionnaire appropriée. Après le retour de fonction du gestionnaire, le contexte est restauré à l'état précédant l'arrivée de l'exception ou de l'interruption et le contrôle est retourné au programme à l'endroit où il avait été interrompu. La différence majeure entre les interruptions et les exceptions est que le gestionnaire d'exceptions enregistre un évènement du moniteur de santé du système dans tous les cas, alors que le gestionnaire d'interruption doit plutôt appeler un sous-gestionnaire approprié en fonction de la source indiquée par le contrôleur d'interruptions.

Les sources d'interruptions appartenant au noyau sont traitées en mode superviseur et leur implémentation est classique. Cependant, les sources d'interruptions appartenant aux partitions doivent plutôt engendrer un traitement d'interruption virtuel en mode non privilégié. Dans ce cas, le traitement des interruptions est beaucoup plus complexe.

Le cheminement pour le traitement d'une interruption dans XtratuM-PPC suit les étapes suivantes :

1. Sauvegarde du contexte (prologue).
2. Accusation de réception de l'interruption et détermination de la source par la lecture du registre IACK du contrôleur d'interruptions.
3. Transfert vers le gestionnaire d'interruptions en C du noyau XtratuM :
    a) sauvegarde du reste du contexte du processeur ;
    b) préparation d'une structure de donnée de contexte passée au gestionnaire ;
    c) configuration du processeur en mode superviseur avec gestion de mémoire virtuelle ;
    d) appel du gestionnaire d'interruptions en C du noyau XtratuM.
4. Traitement de l'interruption par le gestionnaire en C (`DoPpcIrq`) :
    a) appel du sous-gestionnaire associé à la source ;
    b) gestion de l'horloge et des minuteries pour déterminer si un changement de contexte de partition doit avoir lieu ;
    c) retour au vecteur d'interruption pour le traitement des interruptions virtuelles.
5. Traitement des interruptions virtuelles, si nécessaire. Sinon, on termine normalement en sautant directement à l'étape 6.
    a) détermination du vecteur virtuel à appeler dans la partition en fonction de la source ;
    b) ajout d'une structure de données d'émulation d'interruption sur la pile qui permet à l'hyperappel de retour d'interruption virtuelle (`XM_ppc_rfi`) de restaurer le contexte convenablement ;
    c) saut en mode usager vers l'adresse du vecteur virtuel dans la partition ;

d) la partition continue ici et le contexte sauvegardé demeure sur la pile jusqu'à l'exécution de l'hyperappel `XM_ppc_rfi`. Le saut à l'étape 6 se fait à partir de l'hyperappel. Il peut s'écouler un délai arbitraire avant l'appel à `XM_ppc_rfi` dans la partition, mais cela n'affecte aucunement le traitement des interruptions ultérieures, dont l'interruption de minuterie nécessaire au changement de contexte de partitions.

6. Restauration du contexte du processeur (épilogue).
7. Retour vers le programme à l'endroit interrompu.

On remarque dans ce cheminement qu'une interruption virtuelle suspend le traitement habituel de l'interruption. On tend plutôt la main à un vecteur virtuel dans la partition, qui se charge d'émuler l'interruption en mode usager. Ce changement de flot d'exécution est permis puisqu'on enregistre une structure de données sur la pile qui permet à l'hyperappel de retour d'interruption virtuelle (`XM_ppc_rfi`) de déterminer à quel endroit on devra éventuellement revenir pour restaurer le contexte enregistré initialement. Si les interruptions virtuelles sont désactivées dans la partition, l'interruption n'est pas traitée immédiatement. Elle est plutôt reportée au prochain moment où les interruptions virtuelles sont réactivées par l'hyperappel `XM_enable_irqs`.

5.5.3 Problèmes potentiels

Le traitement des interruptions dans les systèmes IMA pose plusieurs problèmes déjà identifiés ailleurs [41, 57]. En plus de ces problèmes connus, l'implémentation des interruptions dans notre prototype souffre d'un problème identifié tardivement et dont la solution n'a pas encore été implémentée.

En effet, la structure de données d'émulation d'interruption mentionnée plus avant est le maillon faible de notre implémentation. Cette structure de données est enregistrée sur la pile

de la partition active et elle est nécessaire afin de pouvoir retourner vers le code principal à partir des interruptions et exceptions virtuelles. La composante principale de cette structure de données est l'adresse de retour pour la restauration du contexte. Puisque cette adresse est lue par le code de restauration du contexte exécuté en mode superviseur dans le noyau, elle pourrait être victime d'empoisonnement de la pile (« stack tainting », en anglais) par le code utilisateur. Conséquemment, le code des partitions pourrait avoir accès en écriture à cette donnée à travers la pile. Du code malicieux pourrait modifier l'adresse valide sauvegardée sur la pile par le noyau avant d'effectuer un hyperappel de retour d'exception (`XM_ppc_rfi`). Cet hyperappel saute en mode superviseur à l'adresse spécifiée dans la structure de données sur la pile pour terminer adéquatement la restauration du contexte en fonction du type d'interruption ou d'exception enregistré. L'effet serait donc le saut à une adresse arbitraire en mode superviseur. Dans le meilleur cas, une adresse invalide serait employée et le noyau détecterait la faute. Dans le pire cas, l'adresse serait celle d'une routine malveillante dans l'espace mémoire de la partition, qui serait exécutée en mode superviseur. La partition malveillante pourrait ainsi prendre le contrôle entier du système, ce qui causerait une défaillance totale du partitionnement robuste.

La solution la plus simple à ce problème serait d'employer une pile spéciale pour l'enregistrement des structures de données d'émulation d'interruptions, dont l'accès est uniquement permis au noyau. Cette solution n'est cependant pas triviale à mettre en oeuvre. Puisque le problème a été détecté très tard, nous n'avons pas eu le temps de le corriger dans le prototype. Nous avons cependant enregistré un rapport de bogue afin que la résolution soit faite à un moment ultérieur. Ce problème d'escalade de privilège par empoisonnement de pile n'est cependant pas dramatique dans un contexte de recherche où le noyau est exécuté dans un environnement contrôlé. Par contre, si ce genre de problème survivait sans être détecté jusqu'au déploiement dans un noyau de partitionnement robuste commercial, les conséquences pourraient être dramatiques.

5.6 Synchronisation multicoeur

Tous les programmes qui exploitent le parallélisme de plusieurs processeurs doivent synchroniser leurs opérations à certains moments. La synchronisation est nécessaire pour éviter les conditions de courses et les bogues qui peuvent survenir lorsque des évènements importants ne sont pas séquencés adéquatement. La synchronisation est aussi primordiale dans les systèmes monocoeurs traditionnels quand on emploie des mécanismes de temps partagé.

Un exemple de synchronisation est l'utilisation de verrous d'exclusion mutuelle. Ces verrous servent à entourer les régions critiques d'accès à une donnée partagée dans chaque fragment pouvant s'exécuter en concurrence. Un verrou permet de bloquer l'exécution d'un fragment sur un coeur en attendant qu'un autre coeur ait terminé son traitement sur une donnée partagée. L'extrait 5.1 présente un exemple d'utilisation d'un verrou dans le noyau XtratuM. Dans ce cas en particulier, on veut éviter l'accès concurrent à un registre critique du contrôleur d'interruption. Les routines `SpinLockIrqSave` et `SpinUnlockIrqRestore` sont utilisées pour respectivement verrouiller et déverrouiller l'accès à la région critique dans toutes les fonctions qui pourraient partager un accès à la variable `g_p_PIC_irq2vpr`. Si plusieurs coeurs pouvaient accéder à cette variable en même temps, il pourrait se produire une multitude de différents entrelacements d'opérations, qui pourraient ultimement se solder en un état indésiré de la ressource.

Dans les sous-sections qui suivent, nous présentons l'implémentation de la synchronisation multicoeur dans XtratuM-PPC.

5.6.1 Opérations atomiques

Les opérations atomiques permettent de modifier une variable partagée par plusieurs coeurs en s'assurant qu'un seul des coeurs réussisse l'opération et que cette dernière apparaisse aux autres comme ayant été exécutée atomiquement, c'est-à-dire sans interruption. Les opérations

```
static void PIC_enable_irq(xm_u32_t irq)
{
    xm_u32_t intFlags;

    /* !!! CRITICAL SECTION START !!! */
    SpinLockIrqSave(&picLock, intFlags);

    /* Clear the source's VPR[MSK] bit, thus enabling it */
    *g_p_PIC_irq2vpr[irq] &= ~PIC_IVPR_MSK;

    SpinUnlockIrqRestore(&picLock, intFlags);
    /* !!! CRITICAL SECTION END !!! */
}
```

Extrait 5.1 Exemple d'utilisation d'un verrou d'exclusion mutuelle dans XtratuM

atomiques sont nécessaires, car la plupart des modifications de variables se font à l'aide de séquences lecture/modification/écriture (« read/modify/write » ou RMW en anglais). Les processeurs PowerPC ne sont pas dotés d'instructions atomiques réalisant des opérations RMW de manière ininterruptible. On peut considérer les opérations atomiques comme des « microverrous » autour d'une modification simple de variable en mémoire. Puisque les primitives de synchronisation et l'accès aux données partagées reposent souvent sur l'utilisation d'opérations atomiques, nous devons en implémenter un ensemble suffisant. Par exemple, l'opération atomique nommée `XMAtomicIncReturn` dans XtratuM-PPC permet d'incrémenter de manière atomique une variable et il est garanti que cette incrémentation sera unique et perceptible immédiatement par les autres processeurs, peu importe leur hiérarchie mémoire et l'état de leurs caches.

Nous avons implémenté les opérations atomiques autour du squelette fourni initialement dans XtratuM. La version LEON de XtratuM contient en effet un API d'opérations atomiques, mais ces dernières sont réalisées avec des mécanismes non atomiques puisque la véritable atomicité de ces opérations n'était pas nécessaire dans un contexte monocoeur. En analysant le code source du noyau, nous avons déterminé que plusieurs de ces opérations n'étaient même

pas utilisées. Cela nous a permis de redéfinir certaines des opérations de manière opportuniste sans affecter la compatibilité avec le reste de la base de code existante.

L'implémentation d'opérations atomiques sur une architecture donnée est habituellement une tâche difficile étant donné la complexité des interrelations entre le modèle de consistance faible pour la mémoire comme celui du PowerPC et l'effet des mécanismes de cohérence des caches sur le partage de données. Ces problèmes sont déjà bien connus et présentés dans les références déjà mentionnées à la section 2.7 de la revue de littérature. Dans le cas du PowerPC, deux mécanismes doivent être appliqués pour réaliser des opérations atomiques : les barrières et les instructions de lecture et écriture conditionnelles.

Les instructions barrières [2] permettent d'ordonner les instructions d'écriture et de lecture en mémoire, qui autrement auraient pû être exécutées dans un ordre différent de celui spécifié dans le code en raison des caractéristiques de traitement non ordonné du modèle de consistance mémoire faible de l'architecture PowerPC. Elles permettent aussi d'assurer que toutes les instructions émises ont exécuté leurs effets secondaires au point barrière. La barrière de lecture/écriture `sync` a aussi l'effet de synchroniser la visibilité des changements entre les caches des différents coeurs selon le protocole de cohérence des caches. Cette instruction est particulièrement importante, car c'est le seul moyen d'assurer qu'une valeur écrite en mémoire est visible à tous les autres coeurs qui pourraient vouloir y accéder.

Les instructions de lecture et écriture conditionnelles ont la même sémantique que le mécanisme classique d'atomicité par parties « LL/SC » (« Load Linked, Store Conditional ») originaire sur l'architecture MIPS [59, p. 257]. L'instruction PowerPC `lwarx` (« Load word and reserve indexed ») permet d'effectuer une lecture avec réservation d'un bloc de mémoire. Toutes les opérations qui modifient la valeur chargée doivent s'être exécutées avant l'exécu-

2. Les instructions `sync`, `isync` et `eieio` sont respectivement les barrières de lecture/écriture, d'exécution et d'entrée/sorties sur le PowerPC e600.

tion de l'instruction `stwcx.` (« Store word conditional indexed, recording condition »)[3] qui réécrit en mémoire la valeur modifiée. L'instruction `stwcx.` échoue et enregistre un bit de condition si un autre coeur tente une écriture sur le bloc réservé. Le bit de condition peut être vérifié pour réessayer l'opération en boucle jusqu'au succès de l'écriture. L'utilisation judicieuse de ces deux instructions permet d'émuler n'importe quelle opération atomique. Cette catégorie d'instructions est implémentée avec de la logique additionnelle dans le module de cohérence mémoire et dans les caches sur l'architecture e600. Les caches doivent donc être activées et configurées en mode « write-back » pour permettre leur utilisation.

Nous avons basé notre implémentation des opérations atomiques sur celle du port PowerPC du noyau Linux. Cette implémentation est disponible en code source libre et elle est utilisée depuis plusieurs années, en plus d'avoir été révisée par une multitude d'experts. Le fichier original du noyau Linux (version 2.6.33.2) qui contient le code des opérations atomiques est `arch/powerpc/include/asm/atomic.h`. Nous avons adapté ces routines pour respecter l'API original de XtratuM et pour remplacer plusieurs dépendances envers le noyau Linux. Avant d'arrêter notre choix sur l'implémentation Linux, nous avons évalué l'implémentation de U-Boot, mais notre analyse a démontré que celle-ci n'était pas suffisante pour un environnement multicoeur en raison des patrons de barrières employés.

L'extrait 5.2 présente l'implémentation de l'opération atomique d'incrémentation `XMAtomicIncReturn`. On remarque dans l'extrait que l'incrémentation (instruction `addic`) est entourée d'une boucle d'atomicité. La boucle d'atomicité débute avec une lecture (instruction `lwarx`) et se termine par un bloc d'écriture conditionnelle (instructions `stwcx.` et `bne-`) qui force un nouvel essai de l'opération si l'écriture de la valeur modifiée échoue. Avant l'opération atomique, on remarque une barrière mémoire (la macro `SMP_LWSYNC`, équivalente à `sync` dans notre cas) qui force l'exécution complète de toutes les opérations mémoires précé-

3. Il est important de noter que le point final (« . ») dans « `stwcx.` » fait partie de la syntaxe assembleur de l'instruction. Lorsque nous référerons à cette instruction, le point sera inclus. Il ne s'agit donc pas d'une erreur typographique.

dant l'opération atomique. Après l'opération atomique, on retrouve une barrière d'exécution (la macro `SMP_ISYNC`, équivalente à `isync` dans notre cas) qui force l'exécution complète de toutes les instructions dans les queues du processeur, incluant le `stwcx.` final, avant de continuer l'exécution du reste du flux d'instructions.

```
/**
 * Atomically increases the counter and returns the new value
 *
 * @param p_atomic - pointer to an opaque atomic counter
 * @return value of counter after atomic increase
 */
static __inline__ xm_s32_t XMAtomicIncReturn(xmAtomic_t *
   p_atomic)
{
    xm_s32_t tmp;

    __asm__ __volatile__(
    SMP_LWSYNC
"1: lwarx %0,0,%1\n\
    addic %0,%0,1\n\
    stwcx. %0,0,%1 \n\
    bne- 1b"
    SMP_ISYNC
    : "=&r" (tmp)
    : "r" (&p_atomic->counter)
    : "cc", "xer", "memory");

    return tmp;
}
```

Extrait 5.2 Implémentation de l'opération atomique `XMAtomicIncReturn`

Notre librairie d'opérations atomiques complète comporte des opérations arithmétiques et logiques. Les opérations arithmétiques sont utilisées principalement par des compteurs atomiques et dans les barrières. Les opérations logiques sont utilisées principalement dans la gestion des champs de bits liés aux interruptions virtuelles.

5.6.2 Verrous

Les verrous permettent l'exclusion mutuelle de régions critiques pour l'accès à des données partagées. Ils permettent aussi l'implémentation de protocoles de synchronisation pour des séries d'évènements simples.

Comme pour les opérations atomiques, nous avons implémenté les verrous à partir du port PowerPC du noyau Linux. Le noyau Linux comporte une version optimisée de l'algorithme de verrou à reprise (« spinlock » en anglais). Les spinlocks sont conçus spécifiquement pour les multiprocesseurs. L'algorithme est simple : on tourne en boucle sur l'écriture d'une variable verrou partagée jusqu'à ce qu'on ait réussi à la modifier à la valeur verrouillée.

L'extrait 5.3 présente le code du verrou de XtratuM-PPC. Le verrouillage est séparé en deux phases : une phase d'essai de verrouillage (`__ArchSpinTryLock`) qui échoue immédiatement si un autre processeur possède le verrou et une phase de verrouillage bloquante (`__ArchSpinLock`) qui permet de bloquer jusqu'à l'accès à la région critique protégée.

5.6.3 Barrières

En plus des verrous, le noyau XtratuM-PPC fournit une implémentation de barrière qui permet la synchronisation entre les coeurs. Alors que le verrou est utilisé pour sérialiser l'accès à une donnée partagée, un coeur à la fois, la barrière est plutôt utilisée pour faire converger le traitement de plusieurs coeurs au même endroit. Une barrière attend l'arrivée de tous les coeurs avant de les laisser procéder.

Notre implémentation de barrière est une adaptation de l'algorithme classique « sense-reversal barrier » qui permet la réutilisation immédiate de la barrière sans possibilité d'interblocage (« deadlock »). Nous avons adapté le pseudocode de Solihin [59, p.273] pour utiliser les opérations atomiques de XtratuM et pour respecter la syntaxe du langage C.

```
/**
 * Tries to acquire the lock variable pointed to by "p_lock" without
 * spinning.
 *
 * @param p_lock - pointer to lock to acquire
 * @return the old value in the lock, so we succeeded in getting the lock
    if the return value is 0.
 */
static inline xm_u32_t __ArchSpinTryLock(archSpinLock_t *p_lock)
{
    xm_u32_t tmp, token;

    token = 1;
    __asm__ __volatile__(
"1: lwarx %0,0,%2\n /* Read lock in tmp */ \
    cmpwi 0,%0,0\n /* Check if unlocked */ \
    bne- 2f\n /* If locked: return */ \
    stwcx. %1,0,%2\n /* Try to store locking value in lock */ \
    bne- 1b\n /* If store failed, restart atomic section */ \
    isync\n\
2:" : "=&r" (tmp)
    : "r" (token), "r" (&p_lock->lock)
    : "cr0", "memory");

    return tmp;
}

/**
 * Acquire lock, spinning until lock is acquired.
 *
 * @param p_lock - pointer to lock to acquire
 */
static inline void __ArchSpinLock(archSpinLock_t *p_lock)
{
    /* Try to lock until successful */
    while (1) {
        if (__builtin_expect((__ArchSpinTryLock(p_lock) == 0), 1))
            break;
    }
}

/**
 * Unconditionnaly releases lock.
 *
 * @param p_lock - pointer to lock to release
 */
static inline void __ArchSpinUnlock(archSpinLock_t *p_lock)
{

    /* Force barrier before unlocking */
    __asm__ __volatile__("# __ArchSpinUnlock\n\t"
                "sync\n\tisync": : :"memory"); /* Lightweight sync could go
                   here, but this is safe */
    /* Release lock */
    p_lock->lock = 0;
}
```

Extrait 5.3 Implémentation des verrous multicoeurs dans XtratuM-PPC

L'implémentation de la barrière est bloquante et nous n'avons pas fourni de fonction permettant de vérifier l'état d'une barrière, ni ajouté de temporisation, car ces deux opérations complexifient l'algorithme. Les opérations atomiques peuvent cependant être utilisées pour construire des sémaphores, qui à leur tour permettent la réalisation de barrières plus flexibles [23, p.21]. Ces barrières sont cependant moins performantes, car elles requièrent un plus grand nombre d'opérations atomiques, qui chacune affectent le temps d'exécution et les caches des autres processeurs.

5.6.4 Problèmes

L'architecture PowerPC est superscalaire et utilise un modèle de consistance mémoire faible. Les processeurs superscalaires exécutent des instructions dans plusieurs unités de traitement en parallèle pour profiter du parallélisme au niveau des instructions qui n'ont pas d'interdépendances de données. Cette optimisation permet d'augmenter le débit moyen à une valeur supérieure à une instruction par cycle, au détriment d'une incertitude sur le moment réel d'exécution des instructions. En parallèle, le modèle de consistance mémoire faible permet au processeur de réordonner les accès de lecture et d'écriture pour maximiser la possibilité d'atteindre les caches. Le réordonnancement des opérations augmente aussi l'incertitude sur le moment d'exécution des instructions. Ces deux attributs de l'architecture PowerPC permettent aux processeurs qui l'utilisent d'avoir des performances très élevées, au détriment d'une analyse plus difficile.

Les opérations atomiques et les accès à des variables partagées ne sont analysables par le programmeur que si les instructions qui les composent sont exécutées dans le même ordre que celui inscrit dans le code de programme. L'exécution superscalaire et le modèle de consistance mémoire faible impliquent un ordre potentiellement différent de celui inscrit dans le programme. Comme nous l'avons déjà mentionné, l'architecture PowerPC est dotée d'instructions « barrières » qui permettent de forcer un ordonnancement particulier des accès mé-

moire et du flux d'instructions. L'instruction barrière `sync` permet de garantir l'exécution de tous les accès mémoire déjà émis et d'assurer la stabilisation des valeurs selon le modèle de cohérence des caches. Dans les PowerPC multicoeurs, les barrières émettent des messages « broadcast » sur le bus de cohérence mémoire qui forcent les autres coeurs à finaliser leurs accès qui pourraient être en commun avec le processeur ayant exécuté l'instruction barrière.

Considérons un exemple avec deux coeurs, soit C1 et C2. On assume que C2 avait en attente une instruction d'écriture mémoire avec une page ou un bloc en commun avec C1. En exécutant `sync` sur C1, l'exécution sur C1 ne se poursuit qu'après l'exécution de l'instruction d'écriture en attente sur C2. Sans l'utilisation de `sync`, C2 aurait pu attendre un moment plus opportun pour finaliser l'écriture, au détriment d'accès en lecture potentiellement importants par C1. Ces accès en lecture par C1 auraient chargé la valeur antérieure à l'écriture de C2, malgré que l'ordre du programme soit respecté. L'instruction `sync` sur C1 a donc modifié le flux d'exécution de C2, au bénéfice de la logique du code, mais possiblement au détriment du délai d'exécution des instructions sur C2. Ce problème est important si on désire employer un ordonnancement SMP, car dans ces cas la prédictibilité du temps d'exécution peut en souffrir.

En plus de l'effet de perturbation du temps d'exécution entre les coeurs, un autre problème aussi grave découle des barrières de mémoire. En effet, les instructions `sync` et `eieio` émettent des transactions vers le module de cohérence mémoire qui gère l'accès aux ressources partagées. Ces transactions sont de type « address-only » et ne causent aucune transaction extérieure vers les bus de mémoire. Cependant, elles remplissent quand même un emplacement dans la file d'opérations. Si un des coeurs venait à exécuter une boucle continue de `sync`, par exemple en essayant d'acquérir un verrou qui ne sera jamais déverrouillé en raison d'un bogue, il risquerait d'y avoir une quantité énorme de transactions au niveau du module qui gère l'accès aux mémoires externes, ce qui pourrait affamer les autres coeurs ou du moins réduire leurs performances pendant beaucoup plus longtemps qu'un délai d'acquisition de verrou habituel. Une minuterie chien de garde pourrait borner la durée maximale de cette condition, mais nous

n'avons pas implémenté cette solution. Les architectures supportant un modèle de consistance mémoire plus forte, telle que SPARC et certaines versions du ARM, requièrent peu ou pas de barrières pour l'ordonnancement des opérations atomiques. Ces architectures sont donc plus faciles à analyser et souffrent moins du problème de surutilisation du bus mémoire décrit ici.

Finalement, l'utilisation des instructions d'atomicité `lwarx` et `stwcx.` cause l'invalidation de lignes de caches partageant l'adresse physique de la transaction dans les caches L1 et L2 des autres coeurs selon les règles du protocole de cohérence de caches MESI utilisé dans les PowerPC e600 [28]. Ces invalidations sont nécessaires pour assurer la cohérence des caches entre elles. Les variables partagées fréquemment doivent être alignées à la taille d'une ligne de cache dans le but d'éviter que la contention autour du partage ne cause d'invalidations superflues affectant des variables n'ayant aucun lien avec elles. Cette approche cause une certaine sous-utilisation de la mémoire, car la majorité des variables partagées sont de 32 ou 64 bits (1 ou 2 mots de 32 bits) alors qu'une ligne de cache contient 256 bits (8 mots de 32 bits). Cependant, elle permet aussi de réduire les perturbations temporelles causées par des invalidations superflues.

Nous croyons que les problèmes de performances liés à la synchronisation multicoeur ont assez d'influence sur le temps d'exécution du code sur les autres coeurs pour complexifier grandement l'analyse de sûreté des systèmes employant des plans d'exécutions SMP et hybrides. En effet, les outils actuels d'analyse du WCET ne permettent pas une évaluation précise de celui-ci quand des relations de dépendances existent entre les coeurs [21]. Des résultats récents de Fuschen chez SYSGO [30] sur la réalisation SMP d'un noyau de partitionnement robuste suggèrent qu'il est possible d'exécuter des partitions non critiques en mode SMP, mais que les partitions critiques devraient être exécutées dans une fenêtre d'activation où seulement un coeur est actif. Cette restriction permettrait d'au moins éviter tous les effets de partages de caches et de perturbations temporelles qui peuvent découler de l'utilisation simultanée de plusieurs coeurs avec des ressources mémoire partagées. Ce scénario est explicitement supporté

par notre modèle de partitionnement, qui a d'ailleurs été conçu pour permettre l'exécution exclusive sur un seul coeur pendant certaines tranches de temps, comme nous l'avons présenté à la section 4.8.1.

5.7 Mécanisme d'hyperappels

L'implémentation du mécanisme d'hyperappels sur les PowerPC est très différente de la version originale sur LEON. Deux différences architecturales majeures expliquent la réimplémentation complète du mécanisme :

- l'architecture SPARC des processeurs LEON permet de spécifier l'un de 256 vecteurs différents pour les traps et appels systèmes combinés, alors que l'architecture PowerPC ne possède qu'une seule exception liée aux appels systèmes ;
- les systèmes d'exploitation sur PowerPC emploient souvent l'instruction d'appel système `sc` pour les demandes de services en mode superviseur.

L'instruction d'appel système des PowerPC doit pouvoir servir son rôle dans les systèmes d'exploitation de partitions, sans quoi leur implémentation serait complexifiée. Nous avons donc conçu un mécanisme permettant de multiplexer l'instruction d'appel système entre le mécanisme d'hyperappel et l'émission d'une exception virtuelle d'appel système au sein des partitions.

Le mécanisme d'appel système sur PowerPC est le moyen habituel de demander un changement d'état vers le mode superviseur à partir d'un programme en mode usager. Ce mécanisme doit donc servir à la réalisation des hyperappels qui sont obligatoirement des opérations en mode superviseur. En théorie, l'instruction `trap` permet aussi de réaliser un changement d'état vers le mode superviseur, mais le vecteur de cette exception est partagé avec d'autres types d'exceptions de programme et plusieurs instructions sont nécessaires avant d'avoir la certitude d'être en présence d'un « trap » explicite. Pour cette raison, l'instruction `sc` est habituellement privilégiée pour les appels systèmes.

L'instruction d'appel système est utilisée par les routines d'abstractions de `libxm.a` qui effectuent les hyperappels. Cependant, il doit aussi être possible pour un système d'exploitation de partition d'utiliser l'instruction `sc` pour forcer l'exécution transparente d'une exception virtuelle d'appel système sans affecter le reste de l'état du processeur virtuel.

Notre solution à ces deux besoins conflictuels est d'ajouter une clé logicielle autour de l'instruction `sc`. Il est possible d'insérer des données dans le flux d'instructions des PowerPC, pour autant qu'elles ne soient pas exécutées. Avec cette solution, le mot de mémoire suivant l'instruction `sc` doit être soit la valeur zéro pour exécuter un appel système émulé ou une valeur clé particulière pour exécuter un hyperappel. La clé choisie arbitrairement est 0x5abba500. Seuls les 24 bits de poids forts sont validés, alors que les 8 bits de poids faible de la clé servent d'identifiant d'hyperappel si la clé est valide. Pour les hyperappels, la clé doit se retrouver non seulement dans le mot suivant l'instruction `sc`, mais aussi dans le contenu du registre général `r0`. L'implémentation de cette solution est présentée à l'algorithme 5.1. L'extrait 5.4 présente deux exemples d'appels système dans le code assembleur : un hyperappel `XM_ppc_rfi` et un appel système devant être émulé par une exception virtuelle. On remarque dans cet extrait que l'utilisation d'hyperappels et d'appels systèmes émulés ne requiert que peu d'ajouts en comparaison avec l'utilisation native de `sc`, où seule l'instruction elle-même suffit.

Le reste de l'implémentation des hyperappels suit le modèle original du noyau XtratuM, mais avec des conventions adaptées à l'architecture PowerPC.

5.8 Exploitation du MMU

L'aspect le plus complexe de l'implémentation de XtratuM-PPC est l'exploitation du MMU pour l'application du modèle de partitionnement spatial multicoeur. Rappelons que notre modèle requiert l'utilisation d'un MMU pour la protection mémoire gérée par le matériel. Nous avons déjà décrit ce modèle en détail à la section 4.9. La complexité d'implémentation provient du fait qu'un bogue dans l'implémentation du support du MMU engendre habituellement

Algorithme 5.1 Cheminement du traitement des appels systèmes sur PowerPC

```
ExceptionHandler SystemCall()
Save minimal context for initial system call proces-
sing.

    Set processor and MMU state to Kernel Mode.
    Adjust return address to skip key word.
    key := word following sc instruction.
    IF NOT ((key & 0x5abba500) == 0x5abba500 AND
            key == GPR0) THEN

       /* Besoin d'émuler un appel système vir-
       tuel */
       Set proces-
       sor and MMU state to User Mode.
       Fix-up SRR0 and SRR1 regis-
       ters to make it appear like a sys-
       tem call occured.
       Restore minimal context.
       Emulate system call exception.

    ELSE

       /* Traitement d'un hyperappel */
       HypercallNumber := GPR0 & 0x000000FF.
       Process hypercall.
       GPR3 := hypercall return value.
       Process virtual timers.
       Emulate pending traps if necessary.
       Set proces-
       sor and MMU state to User Mode.
       Restore context.

    END IF
Return from exception.
```

des fautes d'accès mémoire irrécupérables. Les causes de ces fautes sont longues à déterminer, car plusieurs interactions non triviales sont impliquées dans la protection mémoire matérielle.

Dans les sous-sections qui suivent, nous présentons la réalisation de la protection de mémoire avec support de MMU matériel. En premier lieu, nous présentons comment le modèle de partitionnement spatial a été appliqué au MMU du coeur PowerPC e600. Ensuite, nous décrivons le

```
/* Hyperappel XM_ppc_rfi */
    /* Chargement de clé dans GPR0. 0xC0 est
     * l'identifiant de l'hyperappel XM_ppc_rfi.
     */
    lis r0,0x5abb
    ori r0,r0,0xa5c0
    /* "System call" suivi de la clé: hyperappel */
    sc
    .long 0x5abba5c0
hyperappel_retourne_ici:
    ...

/* Appel système émulé: pas de clé */
    sc
    .long 0
exception_retourne_ici:
    ...
```

Extrait 5.4 Exemples de code employant l'appel système sous XtratuM-PPC

mécanisme de caches d'entrées du TLB qui a permis une réalisation efficace du changement de contexte de MMU. Enfin, nous discutons des problèmes rencontrés lors de l'implémentation, ainsi que des problèmes de sûreté potentiels liés à l'exploitation du MMU dans un contexte de partitionnement robuste multicoeur.

L'implémentation du support MMU était un élément-clé du développement de XtratuM-PPC. La réalisation de ce support a permis de tester l'exécution du démarrage complet sur un groupe de partitions, ce qui a permis le débogage du reste du noyau. Par la suite, le changement de contexte a été réalisé pour compléter l'implémentation du noyau. L'implémentation du changement de contexte de partition est présentée à la prochaine section.

5.8.1 Application du modèle de partitionnement spatial sur le MMU

Le modèle de partitionnement spatial de notre adaptation multicoeur de XtratuM repose sur l'allocation appropriée de zones de mémoire virtuelle dans le plan de configuration du système. Ces allocations permettent le partage de données entre les groupes de partitions parallèles, ainsi que la protection des données des partitions isolées les unes des autres.

Il existe deux modèles de MMU pour l'architecture PowerPC : le modèle dit « classique » et le modèle « Book E » dédié aux applications embarquées. Ces deux modèles sont décrits en détail dans la définition officielle de l'architecture PowerPC [53]. La différence principale entre les deux est que le modèle « classique » supporte la recherche matérielle de la table des pages, alors que le modèle « Book E » nécessite une implémentation logicielle de cette logique de recherche. La recherche matérielle de la table des pages est problématique dans les applications embarquées pour deux raisons : 1) le délai nécessaire pour la recherche peut être assez long et il est impossible de l'arrêter et 2) l'agencement de la table des pages classique n'est pas flexible.

Les coeurs e600 implémentent un compromis basé sur le modèle « classique » : il est possible d'utiliser un mode logiciel de recherche de table des pages qui contourne la recherche matérielle. Cela permet d'implémenter des algorithmes flexibles de mappage de mémoire virtuelle, comme il est possible de le faire avec le modèle « Book E ». Dans notre cas, nous exploitons le mode logiciel de recherche de table des pages [4] pour éliminer l'indéterminisme temporel causé par la recherche matérielle et pour simplifier la protection de la mémoire.

Deux types de ressources de translation d'adresse sont disponibles dans les coeurs e600. Ces deux ressources sont associées aux blocs à translation directe (BTD) et pages à translation directe (PTD) décrits à la section 4.9.2.

La première ressource, qui réalise les BTD, est nommée « Block Address Translation » (BAT). Les BAT sont implémentés à l'aide d'un TLB à associativité complète à 8 entrées chacune pour les adresses de données et les adresses d'instructions.

La deuxième ressource, qui réalise les PTD, est un système de gestion de mémoire virtuelle segmenté et paginé, exploitant un accès à deux niveaux classique avec des pages de 4 ko et des

4. Le mode logiciel de recherche de table des pages est activé par le bit STEN du registre HID0 (« Hardware Implementation-Dependent register 0 ») lors du démarrage du noyau.

adresses virtuelles de 52 bits. On dit que l'accès est à deux niveaux parce qu'il est nécessaire d'aller chercher dans une table en mémoire (la table des pages) pour obtenir la description du mappage d'une adresse virtuelle à une adresse physique. Il y a donc effectivement deux accès (ou plus) à la mémoire pour chaque transaction effectuée par le processeur à une adresse virtuelle. Les MMUs de données et d'instructions disposent chacun d'un TLB de 128 entrées de deux voies à associativité par jeu pour les accès de pages. Ces TLB permettent d'éviter l'opération de recherche d'entrée de table des pages pour les pages les plus souvent utilisées, ce qui améliore grandement les performances d'accès à la mémoire. Pour que les pages soient à translation directe, comme cela est nécessaire pour le mécanisme de PTD de notre noyau, nous devons employer exclusivement les TLB en configuration statique. Il n'est donc pas permis d'utiliser la recherche matérielle des tables de pages, puisqu'elle est inutile lorsque les TLB sont utilisés en exclusivité.

Nous avons configuré le MMU du coeur e600 pour suivre le flot de translation mémoire suivant :

1. L'adresse effective de 32 bits provenant de l'accès mémoire demandé par le programme est envoyée au MMU.
2. Le MMU détermine si l'adresse effective est couverte par un des BAT (BTD). Si c'est le cas, l'accès est validé en fonction du BAT associé. L'adresse physique est directement obtenue à partir des registres du BAT.
    - Si l'accès mémoire était permis, il est effectué et le traitement prend fin.
    - Si l'accès mémoire n'était pas permis, une exception d'accès interdit est générée.
3. L'adresse effective n'était pas couverte par un BAT. Le coeur génère alors une adresse virtuelle de 52 bits (voir figure 5.6 pour le cheminement) à partir de registres de segments préconfigurés et de l'adresse effective.

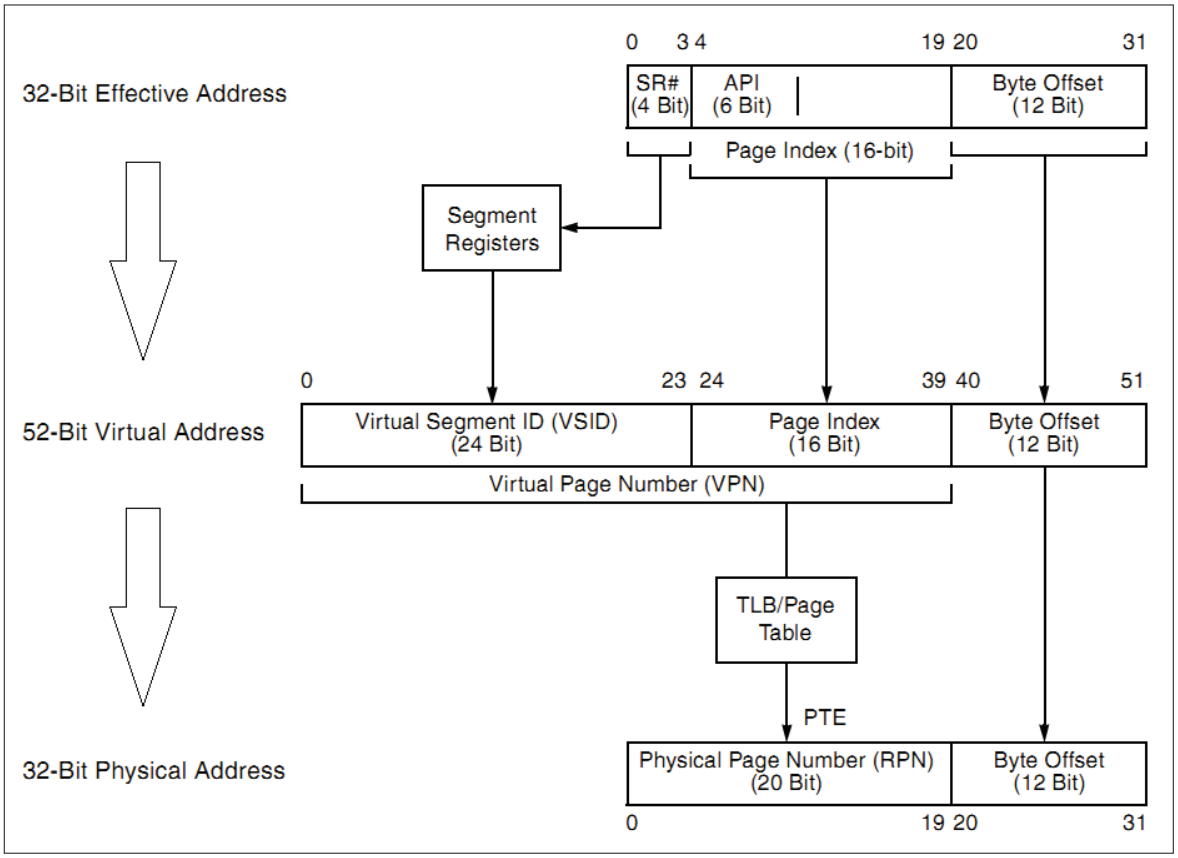


Figure 5.6 Cheminement de la translation d'une adresse effective en adresse virtuelle, tirée de [27, p. 7-28] avec la permission de Freescale Semiconductors

4. L'adresse virtuelle est validée par le TLB de table des pages (PTD). Deux cas peuvent alors survenir :

   a) L'adresse est trouvée dans une entrée de table des pages dans le TLB. L'adresse physique est obtenue à l'aide des données de l'entrée de table des pages et l'accès est validé.

      - Si l'accès mémoire était permis, il est effectué et le traitement prend fin.
      - Si l'accès mémoire n'était pas permis, une exception d'accès interdit est générée.

   b) L'adresse n'est pas trouvée dans une entrée de table des pages dans le TLB. Il s'agit alors automatiquement d'un accès invalide.

- Une exception d'accès interdit est générée.

La translation d'adresse par les BAT est très flexible et couvre la majorité des cas. Elle est aussi plus rapide au niveau du processeur que l'accès à la mémoire à travers les adresses virtuelles du modèle segmenté. La limitation majeure des BAT est qu'ils couvrent des blocs de grande taille et ces derniers doivent être alignés à la frontière de leur taille. Quelques entrées sont réservées par le noyau pour son code, ses données, ainsi que les régions d'entrées/sorties. Les autres entrées de BAT sont assignées automatiquement par le noyau à des BTD du plan de configuration du système. L'assignation est réalisée lors de l'initialisation d'une partition.

La translation d'adresses par le système d'accès mémoire segmenté et paginé est moins flexible, car il n'est possible d'exploiter que 128 pages de données et 128 pages d'instructions, soit un total de 1 Mo en tout, en raison de l'utilisation statique et exclusive du TLB. Dans le cas habituel où les TLB peuvent être rechargés à travers une recherche de table des pages, il est possible d'effectuer le mappage d'une quantité arbitraire de mémoire avec une granularité de 4 ko, au détriment d'un délai d'accès imprévisible et de l'occurrence de plusieurs fautes de pages pour le rechargement des TLB.

L'utilisation réduite de la mémoire virtuelle segmentée et paginée pour l'implémentation des PTD n'est pas un problème en pratique, car chaque partition peut exploiter un grand nombre de BTD. Les BTD couvrent donc la plupart des besoins de mappage de mémoire. Les PTD servent uniquement au mappage de registres de périphériques et aux piles qui sont habituellement de tailles plus faibles que le plus petit BTD disponible à travers les BAT du coeur e600, soit 128 ko.

Rappelons aussi que les conventions de régions d'images de partitions décrites à la section 4.9.2 requièrent l'utilisation d'adresses virtuelles et physiques identiques pour la majorité des régions. L'exploitation d'un espace mémoire virtuel énorme est donc proscrite par ces conventions qui forcent l'utilisation d'adresses à l'intérieur de la zone physiquement im-

plémentée sur la plateforme. La configuration des registres de segments force justement la génération d'adresses virtuelles de 52 bits qui sont toutes physiquement valides lorsqu'elles sont tronquées à 32 bits.

Le MMU du coeur e600 emploie un ensemble de 6 bits pour déterminer le type d'accès mémoire permis pour les BTD et les PTD. Ces bits et leur signification sont présentés au tableau 5.3. Le mappage entre les bits de configuration d'accès et les différents types de zones de mémoire virtuelle est présenté au tableau 5.4. On remarque dans ce dernier tableau que toutes les zones de mémoire sont obligatoirement à cohérence de cache assurée par le matériel, sauf dans le cas des régions d'entrées/sorties. Les conventions d'accès à la mémoire partagée pour les groupes de partitions parallèles dépendent de cette configuration. De plus, les zones de type « code » sont à lecture seule pour les partitions. Il est donc interdit pour les partitions d'employer du code automodifiant. Le mode de lecture seule sur le code permet aussi d'éviter l'écrasement du programme en cas de débordement de tampon à l'intérieur d'une partition.

Tableau 5.3 Bits de configuration d'accès mémoire du MMU du coeur e600

Bit de configuration	**Description**
W	« Write-through », soit une politique de cache à écriture immédiate.
I	« Inhibited », soit un accès contournant la cache.
M	« Memory coherence required », soit un accès à travers la cache dont la cohérence inter-cache est assurée par le matériel.
G	« Guarded », soit un accès qui ne peut être effectué spéculativement ou dans un ordre différent de celui émis.
PP[0]	Lecture permise (si activé)
PP[1]	Écriture interdite (si activé)

Tableau 5.4 Configuration d'accès des différents types de zones de mémoire virtuelle définies dans XtratuM-PPC

Type	PP	WIMG	Description
« code »	01	M	Lecture seule.
« io »	10	IG	Lecture/écriture, contournement de cache, accès spéculatif interdit, pas de cohérence mémoire (puisque la cache est contournée).
« data »	10	M	Lecture/écriture.
« stack »	10	M	Lecture/écriture.

Chaque partition peut exploiter l'ensemble des ressources du MMU pour le mappage de sa mémoire, à l'exception des BAT et entrées de TLB de pages déjà reservés par le noyau. Lors du changement de contexte de partitions, les BAT et les TLB de pages sont vidangés et rechargés avec la configuration nécessaire au mappage de la mémoire de la nouvelle partition. Pour ce faire, une cache d'entrées du TLB précalculée est utilisée. En résumé, l'implémentation du modèle de partitionnement spatial multicoeur décrit à la section 4.9 est réalisée sur les PowerPC e600 de la manière suivante :

- Le modèle de MMU « classique » du PowerPC est employé.
- Le MMU est configuré avec l'option de recherche de table des pages par logiciel pour éviter d'avoir à fournir des tables de pages en mémoire, puisque nous utilisons exclusivement le TLB pour les accès de pages.
- Les BTD sont réalisés à l'aide de 8 BAT de données et 8 BAT d'instructions dont la majorité sont encore disponibles pour chaque partition après la réservation par le noyau.
- Les PTD sont réalisés à l'aide du modèle de mémoire virtuelle segmentée et paginée dont les entrées de TLB sont préconfigurées.
- Les adresses virtuelles et physiques sont identiques dans notre implémentation, ce qui simplifie la gestion des registres de segments et des TLB.

- Les différents types de zones de mémoire virtuelle sont associés à des configurations de privilèges fixes afin de générer les exceptions appropriées automatiquement lors d'un accès invalide.
- Lors du changement de contexte de partition, le MMU est entièrement reconfiguré afin de refléter la configuration de mappage de mémoire de la nouvelle partition.

Dans la prochaine sous-section, nous décrivons les caches d'entrées des TLB qui permettent une réalisation simple et des performances élevées lors de la reconfiguration du MMU du changement de contexte des partitions.

5.8.2 Caches d'entrées des TLB

Lors du changement de contexte de partition, la configuration du MMU pour la partition sortante est vidangée et le MMU est reconfiguré en entier pour supporter la carte mémoire de la nouvelle partition. Le changement de contexte de partitions est décrit en détail à la section 5.9. Nous ne décrivons dans cette section que la composante liée au MMU.

Comme nous l'avons déjà mentionné, il y a 16 entrées de BAT pour les BTD (8 chacune pour les instructions et les données) et 256 entrées de pages dans les TLB (128 chacune pour les instructions et les données). Malheureusement, seul un faible nombre d'entrées de pages sont utilisées par les partitions dans la plupart des cas. De plus, la reconfiguration des registres du MMU nécessite l'utilisation d'instructions spécifiques dont les paramètres sont encodés de manière non triviale et ces dernières diffèrent entre les BAT et TLB, ainsi qu'entre les TLB d'instructions et de données. La configuration du MMU par simple copie de mémoire en bloc est donc impossible. Ces facteurs rendent la configuration du MMU complexe et coûteuse en temps.

En plus de la complexité de chargement, la configuration du MMU doit être réalisée en entier durant le changement de contexte, contrairement aux systèmes d'exploitation traditionnels où

le temps nécessaire est amorti à l'utilisation par l'arrivée de fautes de pages seulement lorsque nécessaire.

Afin de contrer ces problèmes et d'optimiser le temps de changement de contexte, nous avons développé un système de caches d'entrées des TLB qui simplifie la procédure de configuration du MMU, tout en bornant le délai de chargement des données à une valeur proportionnelle au facteur d'utilisation des TLB. Ainsi, le changement de contexte d'une partition exploitant un faible nombre de pages sera beaucoup plus rapide que dans le cas d'une partition qui utilise la totalité des entrées disponibles.

Le concept des caches est simple : des structures de données sont construites lors du chargement des partitions par le noyau et leur format est conçu pour simplifier le chargement des registres et éviter la répétition des calculs. Ces structures de données sont génériques et réutilisables entre les différents types de TLB (BAT et TLB de pages). Puisque les TLB de pages sont associatifs par jeu, certains calculs doivent être effectués avant leur chargement pour déterminer quelle entrée effective d'un TLB doit être chargée. Les adresses physiques et réelles ainsi que les types de zones de mémoire virtuelle doivent aussi être convertis en valeurs compatibles avec les champs des registres des TLB. Ces calculs ne sont réalisés qu'une seule fois, lors de la construction des caches. La structure de données d'une entrée dans les caches d'entrées des TLB est présentée à l'extrait 5.5. On remarque dans cette structure de données trois éléments : un mot de contrôle employé par l'algorithme de chargement des TLB, ainsi que les valeurs brutes précalculées (`valueHi` et `valueLo`) des registres de contrôle des TLB.

En plus de contenir des entrées précalculées, la cache de TLB de pages est comprimée afin de déplacer les 256 entrées éparses vers un bloc pouvant être traité linéairement, sans discontinuité ni recherche. Cette compression permet d'itérer uniquement à travers un nombre d'entrées égal au nombre de pages utilisées au lieu d'avoir à traiter 256 entrées à chaque changement de contexte.

```
typedef struct
{
    /** Control word loaded in a GPR before tlbli or tlbld,
     * or alternatively, data describing the BAT entry. */
    xm_u32_t control;
    /** Value of PTEHI or BATxU for the cache entry */
    xm_u32_t valueHi;
    /** Value of PTELO or BATxL for the cache entry */
    xm_u32_t valueLo;
} tlbCacheEntry_t;
```

Extrait 5.5 Structure de données d'une entrée dans les caches d'entrées des TLB

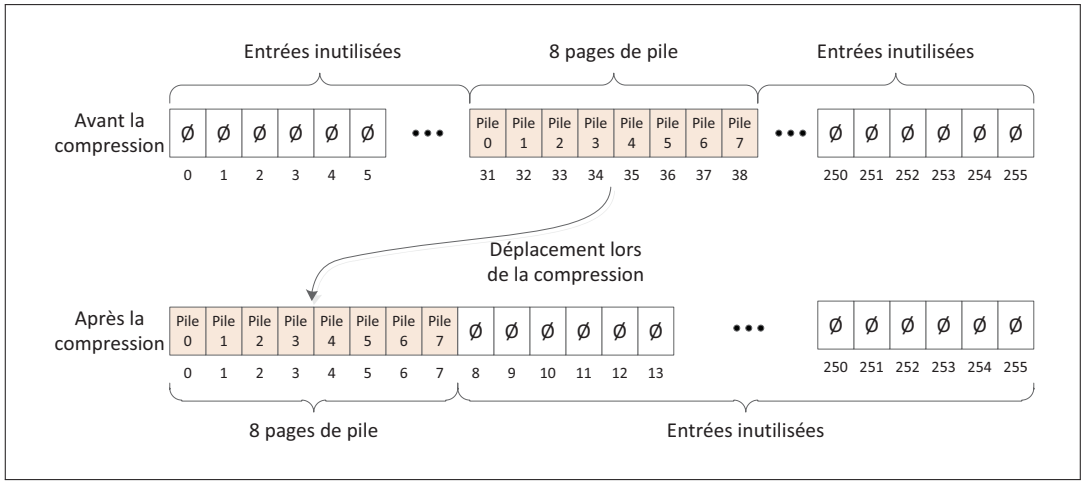


Figure 5.7 Illustration de la compression de la cache d'entrées des TLB de pages

Prenons par exemple le cas d'une partition dotée d'une pile de 32 ko couverte par des PTD et dont le reste des données sont couvertes par des BTD. Ce scénario équivaut à un taux d'utilisation de 3 % des TLB de pages : les 32 ko de la pile sont couverts par 8 pages (PTD) de 4 ko, à partir de 256 disponibles dans les TLB. Cet exemple est illustré graphiquement à la figure 5.7. Sans compression, l'ensemble des 256 descripteurs de pages de la cache devrait être itéré lors du changement de contexte pour assurer que toutes les entrées ont été couvertes par la recherche. Avec la compression, ces 8 entrées se retrouvent au début du tableau de cache et peuvent être chargées en 8 lectures de descripteurs, sans recherche.

En plus de réduire le nombre d'instructions nécessaires, la compression de la cache de TLB de pages est optimale en utilisation de la cache de données du processeur, car la localité des données est maximale. La compression ne permet cependant pas de réduire l'utilisation en mémoire, car l'allocation est effectuée statiquement lors du démarrage afin d'assurer un espace suffisant au cas où les 256 entrées de pages seraient utilisées. L'importance de cette optimisation est mise en relief dans notre étude de cas à la section 5.10.

5.8.3 Problèmes

Lors de l'implémentation du support du MMU dans XtratuM-PPC, nous avons rencontré plusieurs problèmes. Le débogage de cette composante du noyau a nécessité plus de temps que n'importe quel autre module. En plus des problèmes d'implémentation, nous avons identifié deux problèmes de sûreté potentiels, dont un problème critique.

Au niveau de l'implémentation, nous avons rencontré deux problèmes principaux qui ont chacun été assez longs à corriger. La visibilité des registres internes du MMU offerte par la plateforme virtuelle Simics a grandement aidé à la découverte des bogues sous-jacents aux problèmes rencontrés.

En premier lieu, nous avons eu beaucoup de difficulté à déterminer la cause d'exceptions de MMU imprévues. Après le traçage complet du changement de contexte de MMU sur la plateforme virtuelle Simics, nous avons déterminé que le mode de gestion logicielle des tables de pages causait une exception de MMU lors de la première écriture à une page dont l'entrée de table des pages n'était pas enregistrée comme « changée », afin que ce bit de statut puisse être mis à jour par logiciel dans l'exception. Nous avions assumé que la valeur de ce bit n'aurait aucun effet si nous utilisions uniquement les TLB sans support de recherche de tables de pages par le matériel. En raison de cette supposition invalide, nous avions configuré une valeur « jamais changée » à toutes les entrées des TLB de pages, ce qui causait une faute de page imprévue lors de la première écriture à une page. La modification de la valeur par défaut

a éliminé ces exceptions superflues qui causaient des délais temporels, en plus de nécessiter un traitement spécial dans le gestionnaire d'exceptions du MMU.

En second lieu, nous avions omis une information importante dans la configuration du mot de contrôle des caches d'entrées des TLB de pages. L'information manquante forçait la mise à jour d'une seule entrée de TLB, peu importe laquelle devait être mise à jour. Cela engendrait des fautes d'accès de MMU à chaque fois que des pages étaient accédées, puisque les TLB se trouvaient à ne contenir qu'une seule entrée, l'entrée 0, qui était invalide pour tous les accès. Le bogue a été trouvé grâce aux outils de visualisation du TLB de la plateforme virtuelle Simics. Un traçage du code de configuration a ensuite permis de déterminer la source de l'information manquante. La spécification écrite dans nos notes comprenait l'étape de configuration, mais cette dernière était manquante dans l'implémentation du code.

En ce qui a trait aux problèmes de sûreté, nous avons identifié deux problèmes majeurs. Le premier problème de sûreté est lié au changement de contexte du MMU dans un environnement multicoeur. L'autre problème est l'absence de revalidation des données de configuration des zones de mémoire virtuelles lors du chargement du noyau, qui permettrait l'application d'une configuration invalide ou dangereuse lors de l'exécution du partitionnement.

Le premier problème est une conséquence de l'architecture des coeurs e600. Dans cette architecture, l'instruction de vidange d'entrées de TLB « `tlbie` » force la diffusion d'une transaction globale de vidange vers le module de cohérence mémoire. Lorsque la diffusion de ces messages est permise, les coeurs qui observent le bus de cohérence doivent aussi vidanger leur entrée de TLB associée lorsque survient l'occurrence du message global. Dans notre cas, cela équivaudrait à une violation du partitionnement robuste. L'isolation spatiale serait affectée puisqu'un coeur pourrait forcer une invalidation de TLB qui pourrait encore être nécessaire par l'autre coeur. L'isolation temporelle serait elle aussi affectée puisque l'invalidation prend un temps non nul et survient de manière asynchrone et imprévisible. Nous avons déjà mentionné

que chaque coeur exécute une instance locale du noyau et que les horloges locales peuvent être désynchronisées. Puisque nous ne forçons pas un changement de contexte global à tous les coeurs, les suppositions architecturales du modèle SMP à la base de cette caractéristique matérielle sont invalides dans notre application. La solution à ce problème est la désactivation temporaire de la diffusion des messages de cohérence mémoire liés aux opérations de cache (bit « ABE » du registre HID1) lors du changement de contexte du MMU. Cette désactivation doit être temporaire, car d'autres opérations liées aux caches cohérentes dépendent du même bit de configuration (HID1[ABE]). Cette solution semble fonctionnelle dans notre implémentation sur plateforme virtuelle. Cependant, une validation sur le matériel devrait être réalisée, car il est difficile de déterminer tous les effets secondaires possibles de cette mesure. Les manuels de l'architecture e600 sont avares de détails sur cette possibilité et toutes les notes qu'ils contiennent spécifient que le bit HID1[ABE] doit toujours être activé dans les systèmes multiprocesseurs. Notons qu'un bogue d'implémentation de la plateforme virtuelle Simics causait justement la diffusion inconditionnelle des invalidations de TLB, même lorsque le bit HID1[ABE] était désactivé. Nous avons implémenté une solution temporaire par scriptage pour que le comportement réalisé par l'instruction `tlbie` sous Simics respecte les spécifications.

Le deuxième problème de sûreté relevé au niveau de l'exploitation du MMU concerne la validation des données lors du chargement des partitions. Lorsque le plan de configuration du système (PCS) est compilé en table de configuration binaire par l'outil `xmcparser`, la configuration des zones de mémoire virtuelle est valide dans la table résultante. Cependant, si la table de configuration binaire est altérée, soit par des corruptions de mémoire dans le système cible, soit par un agresseur dans la chaîne d'évènements du processus de déploiement, la sûreté du système entier peut être compromise. Cela est dû au fait que notre prototype de XtratuM-PPC n'inclue aucune validation des contraintes du PCS au moment du chargement des partitions. On assume donc que toutes les données sont valides. Puisque les zones de mé-

moire décrites dans le PCS affectent directement la configuration du MMU, il est possible que des bris d'isolation spatiale soient introduits lorsque des configurations erronées ou imprévues sont employées. Plus généralement, tous les noyaux de partitionnement robustes sont à risque d'être affectés par ce problème de configuration erronée du MMU si leurs tables de configuration sont altérées dans un contexte de confiance. Une solution à ce problème de sûreté grave serait d'employer des signatures de validation calculées à partir de la version de confiance de la table de configuration binaire. Ainsi, toute altération pourrait être détectée lors du démarrage du système et une action corrective pourrait être exécutée.

5.9 Changement de contexte de partitions

Après l'implémentation du support MMU, décrit à la section précédente, il était enfin possible de tester l'exécution d'un système à une seule partition. L'implémentation du changement de contexte de partitions a complété la réalisation de la version adaptée de XtratuM-PPC en permettant l'exécution de systèmes complets avec des plans d'exécution arbitraires. Dans les sous-sections qui suivent, nous présentons les détails d'implémentation du changement de contexte de partitions sur les PowerPC.

5.9.1 Appel de l'ordonnanceur

L'ordonnancement des partitions dans XtratuM est effectué par l'appel de la fonction `Scheduling()` du noyau. L'appel de la fonction `Scheduling()` est l'élément central de l'implémentation du partitionnement temporel, car c'est cette fonction qui exécute l'ordonnanceur statique et qui effectue le changement de contexte, lorsque nécessaire. Cette fonction n'est disponible qu'à l'intérieur du noyau. Elle est appelée par l'interruption de minuterie locale à la fin de chaque fenêtre d'activation. Elle est aussi appelée à chaque fois qu'une exception ou qu'un hyperappel modifient le statut de la partition active. Puisque l'ordonnancement et le changement de contexte ne sont pas des opérations réentrantes, le cheminement de `Scheduling()`

est effectué dans une région critique atomique du noyau durant laquelle les interruptions sont désactivées.

L'exécution de la fonction `Scheduling()` appelle l'ordonnanceur statique qui détermine quelle partition devrait être activée parmi toutes celles qui sont disponibles. Si aucune partition n'est prête, comme dans le cas de la suspension de la partition active avant la fin de sa fenêtre d'activation, la partition par défaut du noyau local est choisie. Cette partition par défaut contient simplement une boucle de perte de temps en attente de la prochaine fenêtre d'activation. La nouvelle partition choisie est ensuite activée par une procédure de changement de contexte de partition. Les prochaines sous-sections décrivent les différentes composantes de cette procédure de changement de contexte.

5.9.2 Composante générique du changement de contexte

L'extrait 5.6 présente le code de la composante générique du changement de contexte de partition dans la fonction `Scheduling()`. Le code est identique pour toutes les architectures, incluant les PowerPC. Les fonctions suivantes, utilisées dans l'extrait, contiennent l'implémentation propre au PowerPC :

- *SwitchKThreadArchPre() :* Préparation de la partition sortante avant le changement de contexte. Cette préparation inclut la sauvegarde des registres spéciaux, tels que ceux des unités de calcul à point flottant et vectorielle (Altivec), ainsi que la désactivation des sources d'interruptions de la partition sortante.
- *CONTEXT_SWITCH() :* Changement de contexte principal entre la partition sortante et la nouvelle partition entrante, tel que déterminé par l'ordonnanceur. Le changement de contexte principal comprend l'état des registres à usage général, les registres de statut du processeur et la configuration complète du MMU.
- *SwitchKThreadArchPost() :* Préparation de la partition entrante après le changement de contexte. La préparation de la partition entrante est pratiquement le contraire du che-

minement de `SwitchKThreadArchPre()`. C'est aussi ici que s'effectue la gestion des caches, telle que leur vidange, lors du changement de contexte.

```
/* _______ CONTEXT SWITCH-OUT STARTS ________ */
/* Arch-specific pre-context-switch hook */
SwitchKThreadArchPre(newK, sched->cKThread);

/* Suspend virtual clocks for current partition */
if (sched->cKThread->ctrl.g)
{
    StopVClock(&sched->cKThread->ctrl.g->vClock,
               &sched->cKThread->ctrl.g->vTimer);
}

/* Run actual arch-specific context switch
 * NOTE:
 * - newK is a pointer to the partition descriptor of the new
 * partition to switch-in as determined by the scheduler.
 * - &sched->cKThread is a pointer to the pointer of
 * the currently running partition's descriptor.
 */
CONTEXT_SWITCH(newK, &sched->cKThread);

/* _______ CONTEXT SWITCH-IN STARTS _______ */
/* Resume the virtual clocks of the new partition */
if (sched->cKThread->ctrl.g)
{
    ResumeVClock(&sched->cKThread->ctrl.g->vClock,
                 &sched->cKThread->ctrl.g->vTimer);
}

/* Arch-specific post-context-switch hook */
SwitchKThreadArchPost(sched->cKThread);
```

Extrait 5.6 Composante générique du changement de contexte dans `Scheduling()`

On remarque dans l'extrait 5.6 que seule la gestion des horloges et minuteries virtuelles est assez générique pour être directement réalisée dans le corps de la fonction `Scheduling()`. Tout le reste du changement de contexte de partition doit être effectué dans les fonctions spécifiques que nous venons de décrire.

5.9.3 Implémentation du changement de contexte principal sur e600

La fonction de changement de contexte principal `CONTEXT_SWITCH()` est la seule à être entièrement inconditionnelle dans le changement de contexte de partition. Il est toujours nécessaire de sauvegarder les registres essentiels du coeur et de son MMU, peu importe la configuration des caches, de l'unité de calcul à point flottant et des autres composantes du processeur.

Dans le cas du coeur PowerPC e600, les registres essentiels pour la sauvegarde sont présentés au tableau 5.5. Sur les 32 registres généraux (`r0` à `r31`), on remarque que les registres `r0` et `r3` à `r12` ne sont pas sauvegardés. Dans le cas du changement de contexte de partition de XtratuM, il s'agit d'une optimisation valable. Les raisons de l'omission sont les suivantes :

- les registres `r0` et `r3` à `r12` sont volatiles, c'est-à-dire que le compilateur assume que leur valeur peut avoir changé entre un appel de fonction et son retour ;
- la fonction `Scheduling()`, qui effectue le changement de contexte, n'a besoin que de respecter le minimum des conventions d'appel du langage C, car elle est toujours appelée de manière synchrone, dans un contexte C valide ;
- même lorsque `Scheduling()` est appelée à partir de l'interruption de minuterie, l'appel est synchrone et respecte les conventions d'appel ;
- lorsque `CONTEXT_SWITCH()` retourne, le contexte peut être celui d'une nouvelle partition, mais le flot d'exécution est tout de même un retour à `Scheduling()` tout juste après le dernier appel de `CONTEXT_SWITCH()` par la partition entrante, la dernière fois qu'elle avait été suspendue.

Les étapes réalisées par notre implémentation du changement de contexte dans `CONTEXT_SWITCH()` sont les suivantes :

1. Sauvegarde des registres listés au tableau 5.5 sur la pile de la partition sortante.

Tableau 5.5 Registres sauvegardés lors du changement de contexte principal sur e600

Registres	Description
XER, CR, LR, CCR	Registres d'état du processeur
r1	Pointeur de pile
r2 et r13	Pointeurs de zones de constantes
r14 à r31	Registres généraux non volatiles

2. Sauvegarde d'un « bloc de continuation » sur la pile de la partition active afin de diriger l'exécution de la prochaine restauration de contexte de cette partition.
3. Échange du pointeur de pile entre la partition sortante et la partition entrante.
4. Chargement de la configuration du MMU pour la partition entrante, tel que décrit à la section 5.8.
5. Extraction du bloc de continuation à partir de la pile de la partition entrante. Deux possibilités existent pour ce bloc : il peut avoir été sauvegardé à l'étape 2 ci-haut, lors du dernier changement de contexte de la partition entrante, ou il peut avoir été manuellement inséré par le noyau lors de l'initialisation des partitions.
6. Saut vers l'adresse décrite dans le bloc de continuation. Cette adresse pointe soit vers une routine d'initialisation, soit vers l'étape 7.
7. Destruction du bloc de continuation sur la pile.
8. Restauration des registres listés au tableau 5.5 à partir de la pile de la partition entrante.
9. Retour vers `Scheduling()`.

Le cheminement de `CONTEXT_SWITCH()` est classique, à l'exception de l'utilisation du « bloc de continuation ». Dans le cas habituel, on restaure le contexte de la partition entrante à partir de l'état sauvegardé la dernière fois qu'elle était une partition sortante. Cependant, lors du premier changement de contexte vers une partition, aucun contexte n'a encore été sauvegardé sur la pile. Il n'y a donc pas de données qui peuvent être restaurées. La gestion habituelle de ce cas est de préenregistrer un contexte bidon dans l'espace mémoire de la pile

de toutes les tâches lors de l'initialisation, en s'assurant que ce contexte causera un retour vers le point d'entrée du code de la tâche lors du premier changement de contexte.

Dans notre cas, cette approche n'est pas suffisante, puisque le démarrage d'une partition équivaut au démarrage d'une machine virtuelle. Il est donc nécessaire d'émuler le démarrage de la machine lors du premier changement de contexte entrant. Le bloc de continuation est la version XtratuM du contexte bidon. Au lieu d'enregistrer un contexte bidon complet, on enregistre plutôt une structure de données qui force l'exécution du bon cas : le chargement du contexte de la partition entrante ou un saut vers l'initialisation de la machine virtuelle de partition si cette dernière vient d'être démarrée. Ces deux cas sont illustrés à la figure 5.8. Après l'initialisation de la machine virtuelle, le premier changement de contexte sortant sera effectué normalement à partir de `Scheduling` et la partition enregistrera alors le bloc de continuation normal.

5.9.4 Gestion des registres spécialisés

Les différentes architectures de processeurs PowerPC comportent chacune une série de registres spécialisés pour les fonctions avancées du processeur, comme l'unité de calcul vectoriel (SPE dans les e500, Altivec dans les e600) et l'unité de calcul à point flottant. Puisque ces fonctions sont optionnelles et qu'il est possible de les désactiver au niveau de chaque partition, ces registres ne sont pas sauvegardés automatiquement lors du changement de contexte. La sauvegarde des registres spécialisés est très coûteuse en temps et en espace mémoire. Par exemple, le FPU standard des PowerPC, tel qu'implémenté dans le coeur e600, comporte 32 registres de 64 bits. L'unité vectorielle Altivec, aussi disponible sur le coeur e600, comporte quant à elle 32 registres de 128 bits, soit une quantité de données quatre fois plus grande que tous les registres généraux (`r0–r31`) du processeur. De plus, chacun de ces registres doit être sauvegardé avec des instructions particulières.

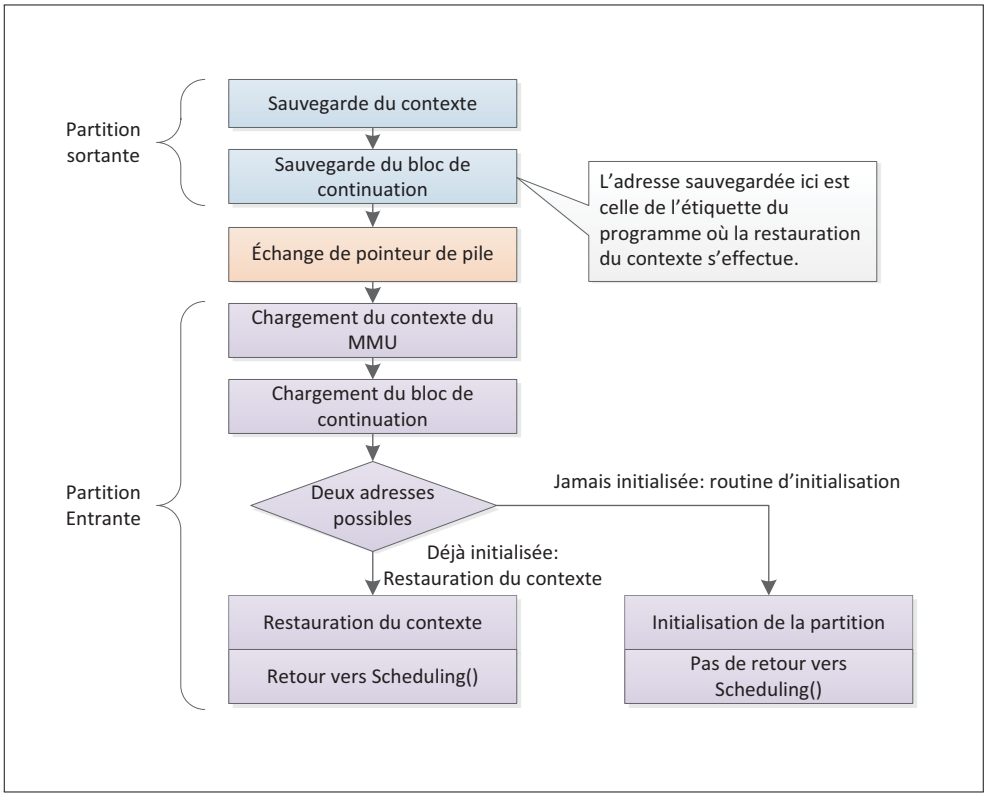


Figure 5.8 Illustration de l'utilisation du bloc de continuation dans le changement de contexte

Afin de simplifier le changement de contexte des registres spécialisés, nous avons séparé leur traitement en phases indépendantes du changement de contexte principal. Ainsi, ces changements de contexte ont été insérés dans les fonctions de préparation de partition sortante et entrante (`SwitchKThreadArchPre()` et `SwitchKThreadArchPost()`). L'espace mémoire pour ces changements de contexte est assigné dans une zone réservée des structures de données de description des partitions au niveau du noyau. L'espace est donc assigné statiquement dans le noyau et ne surcharge pas la pile de chaque partition inutilement. Finalement, l'exécution du changement de contexte de chaque fonction avancée est effectuée conditionnellement à son utilisation par une partition, au lieu d'être exécutée *de facto*, à chaque changement de contexte de partition.

5.10 Mise à l'essai

Dans cette section, nous présentons une étude de cas de système IMA multicoeur exploitant XtratuM-PPC. Cette étude de cas complémente les tests d'intégration continue réalisés lors du développement. Notre étude de cas exécute des interactions représentatives d'un système réel, avec un plan d'exécution multicoeur hybride. Toutes les catégories de fonctions fournies par le noyau XtratuM sont testées, dont les suivantes :

- la gestion virtuelle des coeurs ;
- le changement de contexte complet (processeur, MMU, interruptions, FPU) ;
- les interruptions virtuelles ;
- la communication inter-partitions ;
- le traçage d'évènements du moniteur de santé du système.

L'étude de cas peut être considérée comme un test de stress de l'implémentation, car nous mettons à l'épreuve toutes les fonctionnalités et adaptations réalisées dans le cadre du développement du prototype, en assumant une implémentation complète. Ce test de stress a d'ailleurs mis au jour une dizaine de bogues du noyau, que nous avons dû régler avant de pouvoir obtenir les mesures désirées.

5.10.1 Objectifs

L'étude de cas visait les objectifs suivants :

1. Démontrer que l'adaptation du noyau était fonctionnelle et complète.
2. Fournir un exemple représentatif d'utilisation du noyau, pouvant servir de référence aux autres chercheurs.
3. Permettre de valider expérimentalement certaines hypothèses de performances.

Ces objectifs sont directement liés aux objectifs de recherche principaux énoncés à la section 1.2. Lors de la réalisation de l'adaptation multicoeur de XtratuM, nous avons émis les hypothèses suivantes, que nous voulions initialement valider dans le cadre de nos travaux :

- *Hypothèse 1 :* Étant donné l'utilisation d'une plate-forme virtuelle déterministe sans modèle de cache, les délais de changement de contexte auront toujours les mêmes valeurs dans les mêmes conditions, sans aucune gigue temporelle.
- *Hypothèse 2 :* Le délai de changement de contexte de partition sera proportionnel au nombre de PTD utilisés par une partition.
- *Hypothèse 3 :* Chaque fenêtre d'activation de partition sera au moins aussi longue que la durée spécifiée dans le plan d'exécution, au meilleur de la résolution de l'horloge du noyau.

Lors de la présentation des résultats à la section 5.10.4, ces trois hypothèses seront traitées individuellement.

5.10.2 Présentation de l'étude de cas

Notre étude de cas se veut représentative d'un système IMA. Ainsi, nous avons choisi les applications afin de représenter une combinaison de fonctions commune dans ces systèmes.

La plateforme cible est un système doté d'un processeur PowerPC MPC8461D à trois coeurs cadencés à 792 MHz. La mémoire vive est d'une taille de 512 Mo.

Le noyau XtratuM-PPC est configuré avec un plan d'exécution hybride comportant 4 applications séparées en 7 partitions, dont deux partitions individuelles et deux groupes de partitions SMP. Le tableau 5.6 présente un sommaire des applications. Le nom d'une partition est formé du nom de l'application couplé à l'identifiant du coeur sur lequel elle est exécutée. Par exemple, la partition LZ2 est l'application « LZ » exécutée sur le coeur 2.

Tableau 5.6 Sommaire des applications de l'étude de cas

Nom	Partitions	Description
FFT	FFT0, FFT1, FFT2	Partition de calcul de FFT complexe de 1024 points en point flottant à l'aide de la librairie `kiss-fft` [4]. Réception de blocs à traiter et transmission de blocs traités par la partition FFT0. Calcul de FFT en parallèle dans les partitions FFT1 et FFT2. La partition IO0 est la source et la destination des blocs.
IVHM	IVHM0	Partition de superviseur de sûreté (« Integrated Vehicle Health Monitoring »). Observe en permanence la trace d'évènements de la partition LZ2 et affiche les résultats sur une console virtuelle.
IO	IO0	Partition de serveur d'entrées/sorties : génère des données pour l'application FFT et l'application LZ. Réception et transmission des données en mode « polling » dans des ports de communication à file d'attente. Cette partition est considérée « critique » au niveau du déterminisme temporel. Elle est donc assignée à une fenêtre d'activation temporelle exclusive avec tous les autres coeurs désactivés.
LZ	LZ1, LZ2	Partition de compression/décompression de données avec l'algorithme LZ77 à l'aide de la libraire `bcl` [5]. La partition LZ1 obtient des blocs à partir d'IO0, les traite et renvoie les résultats à IO0. La partition LZ2 génère en parallèle des évènements de trace pour la partition IVHM à l'aide d'une interruption virtuelle d'évènement du noyau. Les deux fils d'exécution de l'application LZ ne communiquent pas entre eux après le démarrage, mais cela leur serait possible.

Les applications FFT et LZ sont représentatives d'applications de calcul et de traitement de données, respectivement. L'application FFT exploite le FPU du processeur. L'application IO est représentative du modèle de serveur d'entrées/sorties exploité en IMA. L'application IVHM est simple, mais représentative du traitement de défaillances et d'évènements systèmes par une partition extérieure.

Les flux de données entre les partitions sont présentés à la figure 5.9. On remarque dans la figure que la partition IO0 communique avec les partitions FFT0 et LZ1 à l'aide de ports de

communication inter-partitions à file d'attente. La taille des blocs transférés est de 8 ko dans tous les cas.

La partition FFT0 aiguille les blocs à destination et en provenance de FFT1 et FFT2 à l'aide de files concurrentes à mémoire partagées. La file développée est à verrouillage grossier et gère la contention en mode FIFO. Les partitions FFT1 et FFT2 essaient de consommer des blocs en concurrence et renvoient leurs résultats à FFT0, qui retourne les blocs à IO0. La partition FFT0 est donc une partition de communication alors que FFT1 et FFT2 sont des partitions de traitement.

La partition LZ1 est similaire à FFT0, mais elle effectue elle-même le traitement des blocs au lieu de déléguer à d'autres coeurs. Le deuxième coeur de l'application LZ, soit LZ2, ne fait que transmettre des évènements de trace du moniteur de santé du système à la partition IVHM0 à travers les mécanismes de trace intégrés à XtratuM.

L'allocation de mémoire des applications est présentée au tableau 5.7. L'allocation temporelle des applications est présentée au tableau 5.8. Le plan d'exécution, quant à lui, est présenté à la section AI-1 en annexe. Les caractéristiques principales du plan d'exécution sont les suivantes :

- le bloc d'activation temporel majeur est d'une période de 100 ms, avec les dernières 10 ms inoccupées ;
- chaque fenêtre d'activation de partition est suivie d'un temps mort d'une durée arbitraire de 1 ms ;
- l'application FFT est un groupe parallèle sur les trois coeurs ;
- l'application LZ est un groupe parallèle sur deux coeurs seulement ;
- l'application IO est exécutée dans une fenêtre d'activation temporelle « critique » dédiée ;

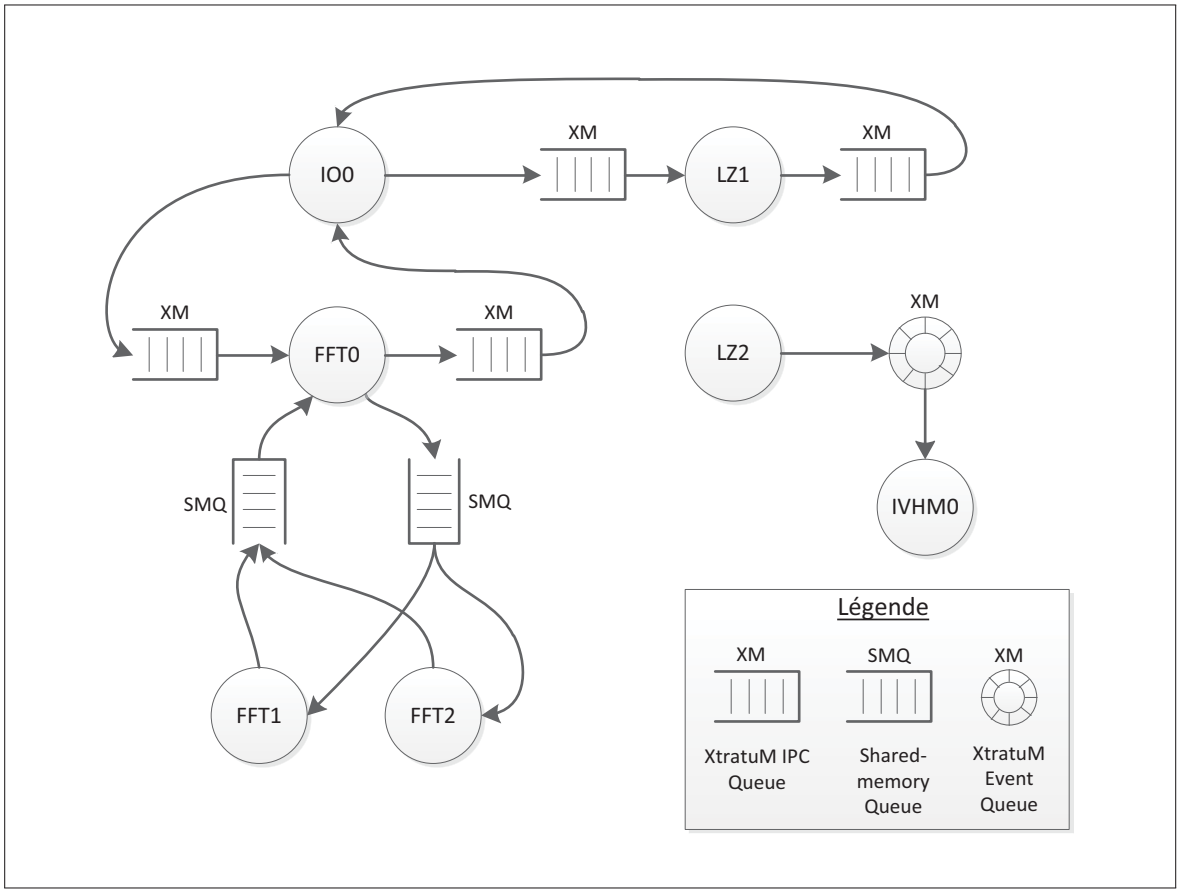


Figure 5.9 Flux de données entre les partitions de l'étude de cas

- la somme de proportion d'utilisation de tous les coeurs est de 166%, c'est-à-dire que le plan d'exécution serait impossible à intégrer sur un seul coeur (plus de 100% d'utilisation).

Nous notons enfin que toutes les fenêtres d'activation d'une même partition sont de la même durée. Cette construction de plan d'exécution est délibérée afin de pouvoir comparer toutes les mesures d'une même partition à une seule valeur nominale.

5.10.3 Méthodologie de mesure

Toutes les composantes de l'étude de cas, soit le noyau, la librairie de services `libxm.a` et les partitions, ont été compilées avec le compilateur GCC 4.5.1, avec optimisation avancée (`-O2`) et symboles de débogage (`-g3`).

Tableau 5.7 Ressources de mémoire allouées aux applications de l'étude de cas

	FFT	**IVHM**	**IO**	**LZ**
Partitions	0, 1, 2	3	4	5, 6
Taille de code	512 ko	128 ko	512 ko	512 ko
Taille de données	128 ko	64 ko	128 ko	128 ko
Taille de pile	128 ko	32 ko	32 ko	32 ko
Taille du bloc de données additionnel	Aucun	Aucun	2 048 ko	Aucun
Nombre de PTD	32	24	8	8
Nombre de BTD	3	2	4	3

Tableau 5.8 Ressources temporelles allouées aux applications de l'étude de cas

	FFT	**IVHM**	**IO**	**LZ**
Partitions	0, 1, 2	3	4	5, 6
Période	50 ms	100 ms	50 ms	25 ms
Temps CPU coeur 0 (ms)	18 ms	8 ms	9 ms	—
Temps CPU coeur 1 (ms)	18 ms	—	—	4 ms
Temps CPU coeur 2 (ms)	18 ms	—	—	4 ms
Utilisation coeur 0 (%)	36 %	8 %	18 %	—
Utilisation coeur 1 (%)	36 %	—	—	16 %
Utilisation coeur 0 (%)	36 %	—	—	16 %

Nous avons effectué les mesures des durées de changement de contexte, de blocs de calcul et de fenêtres d'activation à l'aide de la méthode d'instrumentation VPI. Cette méthode a fait l'objet d'une publication et elle est reproduite à l'annexe II à la page 209. Nous avons donc inséré des appels d'instrumentation « transparents » dans le code du noyau et des partitions. La contribution totale de l'instrumentation au temps d'exécution était de 2 cycles par appel. La méthode de mesure est telle que la contribution finale à la mesure est négligeable (1 cycle par bloc mesuré). Pour chaque partition, nous avons enregistré les mesures suivantes :

- la durée de chaque fenêtre d'activation ;
- la durée du changement de contexte sortant ;
- la durée du changement de contexte entrant ;
- la durée de la composante MMU du changement de contexte.

Nous rappelons que le changement de contexte est réalisé en deux phases pour chaque partition : la phase entrante et la phase sortante. L'illustration sur une ligne du temps de ces deux phases est présentée à la figure 3.10.

Les mesures ont été enregistrées dans un fichier texte lors de l'exécution sur la plateforme virtuelle Simics. Par la suite, nous avons extrait les données du fichier texte à l'aide d'un script Python qui générait un tableau Excel des mesures.

Nous avons mesuré les résultats sur 300 secondes d'exécution virtuelle, pour un total de 3 000 blocs d'activation temporels majeurs et 24 000 fenêtres d'activation sur chaque coeur.

5.10.4 Résultats

Avant de présenter les résultats quantitatifs, nous référons le lecteur aux annexes A I-2 et A I-3, qui présentent respectivement l'enregistrement de la console virtuelle et un extrait du fichier journal enregistré lors de l'exécution. On remarque dans l'extrait du journal d'exécution que les messages informatifs de la console émulée sont juxtaposés avec les résultats numériques de l'instrumentation fournissant les mesures temporelles.

Les résultats des mesures de changement de contexte sont présentés au tableau 5.9. On remarque dans le tableau que pour chaque partition d'une application, la durée du changement de contexte est identique. De plus, les résultats du tableau n'incluent aucune mesure de variance, car toutes les mesures étaient strictement identiques pour une même partition. Nos mesures de durée de changement de contexte valident donc l'hypothèse 1, selon laquelle le changement de contexte requiert toujours le même nombre de cycles, à chaque fois, pour une même partition.

L'absence de variation entre les exécutions individuelles du changement de contexte est la conséquence de deux caractéristiques de l'implémentation. La première caractéristique est l'implémentation du changement de contexte à l'aide d'algorithmes déterministes, dont le

temps d'exécution maximal est borné par une constante selon la configuration statique du système. La deuxième caractéristique est l'exécution sur une plateforme virtuelle parfaite, sans modèle de cache, ni de pipeline, ni de contrôleur de mémoire. Pour cette raison, seule la variabilité due aux algorithmes est prise en compte, sans aucun « bruit » causé par la modélisation d'éléments de performance qui affecteraient le temps d'exécution.

On remarque aussi dans le tableau 5.9 que le changement de contexte entrant est plus lourd que le changement de contexte sortant. Comme nous l'avons déjà mentionné à la section 5.9, le changement du contexte du MMU est coûteux et il est effectué dans la partie entrante du changement de contexte. Le résultat est donc celui qui était attendu. Nos résultats démontrent d'ailleurs que le changement de contexte du MMU consomme plus de 50% du temps total du changement de contexte.

En termes absolus, le changement de contexte de partition coûte moins de 2 microsecondes dans notre étude de cas sur des coeurs cadencés à 792 MHz. Cette performance est acceptable, si on considère que les tranches de temps durent plusieurs millisecondes et que le processeur n'est pas cadencé à sa vitesse maximale de 1.5 GHz. Sur un processeur réel, l'effet des caches et du pipeline se ferait ressentir et ces mesures pourraient gonfler. Nous n'avons pas les modèles nécessaires pour estimer les vrais délais, mais les résultats synthétiques sont encourageants.

Tableau 5.9 Résultats des mesures de changement de contexte

	FFT0	**FFT1**	**FFT2**	**IVHM0**	**IO0**	**LZ1**	**LZ2**
Entrant (cycles)	1097	1097	1097	952	787	746	746
Sortant (cycles)	219	219	219	181	219	181	181
Total (cycles)	1316	1316	1316	1133	1006	927	927
Total (μs)	1.662	1.662	1.662	1.431	1.270	1.170	1.170
MMU seulement (cycles)	880	880	880	774	570	568	568
MMU seulement (μs)	1.111	1.111	1.111	0.977	0.720	0.717	0.717

La figure 5.10 présente le délai de changement de contexte du MMU, ainsi que le nombre de PTD employé, pour chaque partition. L'implémentation du changement de contexte du MMU est faite de telle sorte que le délai d'exécution devrait être directement proportionnel au nombre de PTD utilisé. D'un premier coup d'oeil, il est évident que cette relation existe. Cependant, nous avons dérivé un modèle avec les mesures qui démontre la relation linéaire existante. Dans le graphique de la figure 5.10, les diamants bleus présentent la durée du changement de contexte mesurée, alors que les triangles verts présentent les valeurs de notre modèle. L'équation du modèle est $T_{MMU}(P) = 464 + 13P$, avec P le nombre de PTD d'une partition et T_{MMU} le délai de changement de contexte du MMU en cycles. Le facteur linéaire de 13 dans l'équation est directement relié aux 13 cycles requis pour charger une entrée de PTD dans notre implémentation. L'erreur d'estimation était de moins de 2 cycles dans tous les cas, en raison des effets de bord d'autres composantes du changement de contexte du MMU. Ces résultats valident l'hypothèse 2, selon laquelle la durée du changement de contexte du MMU est directement proportionnelle au nombre de PTD utilisé par une partition.

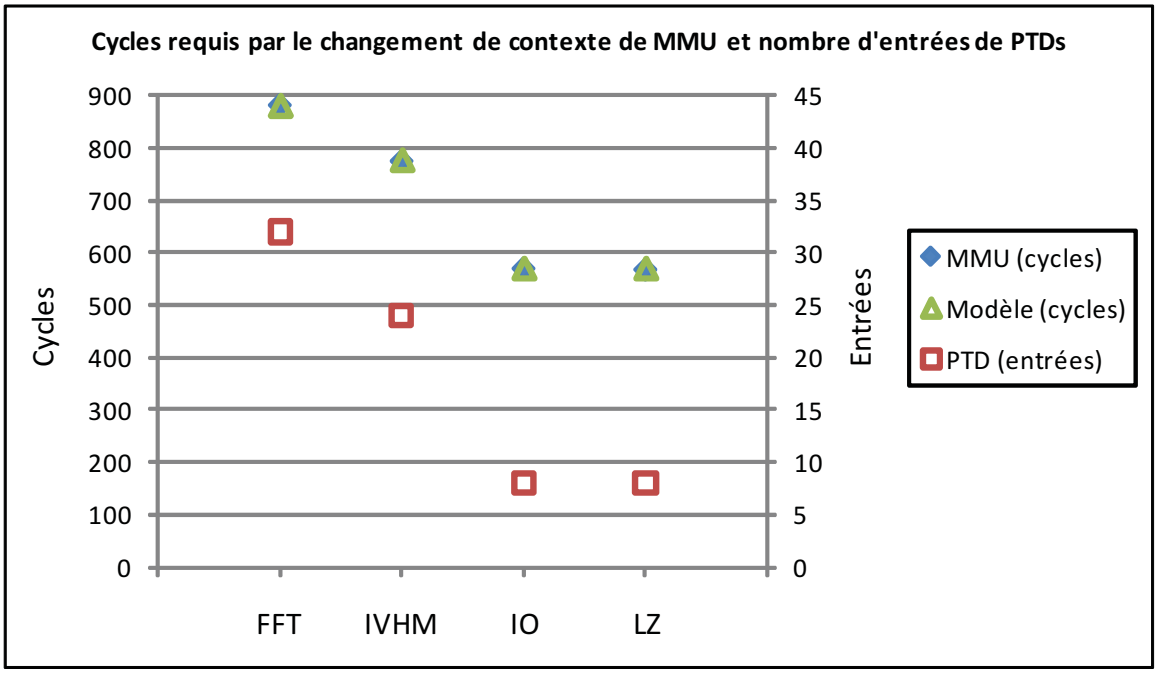


Figure 5.10 Délai de changement de contexte du MMU et nombre d'entrées de PTD

Finalement, le tableau 5.10 présente les résultats des mesures de durée de fenêtres d'activation. Ces durées de fenêtres incluent la contribution du changement de contexte. On remarque dans le tableau que les durées minimales de fenêtres sont toutes à l'intérieur de la tolérance acceptée par rapport à la valeur nominale, validant ainsi l'hypothèse 3. Nous rappelons que la tolérance est d'une microseconde et qu'elle est basée sur l'incertitude de mesure liée à la résolution de l'horloge du noyau. La valeur minimale est importante, car le noyau de partitionnement robuste vise à garantir un temps minimum à chaque partition, sans quoi les échéances temporelles pourraient être manquées. Selon nos mesures, il existe une certaine variabilité, causée par l'exécution d'hyperappels longs tout juste avant l'arrivée de l'interruption de minuterie pour le changement de contexte. Cet effet est décrit en détail à la section 3.7. Cependant, cette variabilité, comme prévu, ne fait qu'allonger les fenêtres d'activation. Nos résultats démontrent que l'implémentation de l'ordonnanceur original de XtratuM et celle de notre gestion d'interruptions et de minuteries adaptée aux PowerPC permettent effectivement de garantir la durée minimale des fenêtres d'activation.

Tableau 5.10 Résultats des mesures de durée de fenêtres d'activation

	FFT0	**FFT1**	**FFT2**	**IVHM0**	**IO0**	**LZ1**	**LZ2**
Nominal (ms)	*9.0000*	*9.0000*	*9.0000*	*4.0000*	*9.0000*	*4.0000*	*4.0000*
Moyenne (ms)	9.0004	9.0001	9.0001	4.0001	9.0003	4.0000	4.0000
Minimum (ms)	*8.9998*	*8.9998*	*8.9999*	*3.9997*	*8.9995*	*3.9994*	*3.9994*
Maximum (ms)	9.0014	9.0009	9.0009	4.0011	9.0009	4.0085	4.0004
Médiane (ms)	9.0002	8.9999	8.9999	3.9997	9.0006	4.0004	4.0004
Écart-type (μs)	451	219	219	181	219	181	181

5.10.5 Conclusions

Le plan d'exécution hybride que nous avons développé met en évidence les caractéristiques nouvelles du modèle de partitionnement temporel de notre adaptation multicoeur de XtratuM. Le taux d'utilisation cumulé de 166% démontre que notre adaptation permet d'exploiter le

parallélisme offert par un processeur multicoeur. XtratuM-PPC permet ainsi d'augmenter le nombre de fonctions intégrées sur un même processeur physique.

Les résultats de notre étude de cas valident aussi nos trois hypothèses de performance. Bien sûr, ces mesures ne font pas le poids d'une preuve formelle de la conception. Cependant, du point de vue du prototypage d'applications IMA et de la recherche dans le domaine des noyaux de partitionnement multicoeur, nos résultats suffisent à démontrer que notre implémentation respecte les suppositions données initialement. Qui plus est, le test de stress à l'encontre du noyau, réalisé dans le cadre de l'étude de cas, confirme l'achèvement de l'implémentation et fournit un exemple complet à partir duquel les utilisateurs potentiels pourront baser leurs travaux.

Enfin, l'exécution de l'étude de cas sur la plateforme virtuelle Simics démontre la pertinence et les avantages de l'utilisation de ce genre d'outils pour le développement et l'évaluation des performances de logiciels embarqués complexes. Nos mesures temporelles n'auraient pas été aussi facilement obtenues sur une plateforme réelle, en raison de l'interférence de mesure habituellement présente lorsque la cible doit s'automesurer. De plus, l'émulation de console et l'instrumentation des fonctions critiques ont permis de valider l'exécution de toutes les composantes de l'étude de cas et du noyau sans affecter le temps d'exécution, ni requérir le partage de ressources matérielles pour la communication des résultats. Cette expérience encourageante va dans la même direction que certaines des conclusions initiales de [43], selon quoi les plateformes virtuelles sont un outil doté d'un potentiel intéressant pour l'évaluation fonctionnelle des logiciels embarqués à haut niveau de sûreté, malgré le niveau de détails incomplet des modèles.

5.11 Récapitulation

Dans ce chapitre, nous avons présenté l'implémentation concrète de l'adaptation multicoeur de XtratuM sur l'architecture PowerPC. Nous avons réalisé le prototype de manière incrémentale en nous basant sur la version LEON originale.

Le développement des pilotes de bas niveau a été relativement simple et rapide. Cependant, l'implémentation du démarrage du système, de l'exploitation du MMU et du changement de contexte de partitions ont nécessité beaucoup d'efforts, en raison de leur complexité. Les difficultés principales que nous avons rencontrées sont les suivantes :

- nous n'avions aucun exemple de fonctionnalité équivalente sur lesquels nous référer lors de l'implémentation ;
- l'architecture du coeur e600 est très complexe, mais la documentation du manufacturier n'est pas rédigée avec un objectif didactique ;
- les spécifications originales du MPC8641D comportaient certaines erreurs ;
- le changement de contexte du MMU et le démarrage du système devaient être programmés en assembleur, ce qui a rendu le débogage et la révision du code plus difficile.

Malgré ces quelques difficultés techniques, nous avons réussi à implémenter le prototype complet de XtratuM-PPC. D'ailleurs, tel que nous l'avons démontré, notre prototype s'est avéré suffisant autant pour déceler les problèmes potentiels de l'adaptation d'un noyau de partitionnement robuste vers un processeur multicoeur que pour permettre le prototypage de systèmes IMA complets sur plateforme virtuelle.

À l'aide d'une étude de cas, nous avons démontré que le prototype est achevé et qu'il permet la mise à l'essai de systèmes IMA d'une ampleur suffisante pour les besoins de la recherche académique. De plus, nous avons validé les trois hypothèses de performance suivantes :

1. Étant donné l'utilisation d'une plate-forme virtuelle déterministe sans modèle de cache, les délais de changement de contexte sont toujours de la même valeur dans les mêmes conditions, sans aucune gigue temporelle.
2. Le délai de changement de contexte de partition est proportionnel au nombre PTD utilisé par une partition.
3. Chaque fenêtre d'activation de partition est au moins aussi longue que la durée spécifiée dans le plan d'exécution, au meilleur de la résolution de l'horloge du noyau.

Lors de la réalisation du noyau, nous avons décelé une multitude de problèmes au niveau de la sûreté et du déterminisme temporel. Un résumé des problèmes identifiés est présenté au tableau 5.11.

Les problèmes relevés dans le présent chapitre n'étaient pas tous pressentis lors de la phase d'analyse de l'adaptation présentée au chapitre 4. Plusieurs des problèmes découlent d'ailleurs de choix d'implémentation et de caractéristiques matérielles propres à la plateforme cible, plutôt que de l'architecture elle-même.

Finalement, parmi tous les problèmes d'implémentation décelés, ceux liés à l'effet de la synchronisation multicoeur sur les performances sont les plus importants. En effet, ceux-ci ont assez d'influence sur le temps d'exécution du code pour grandement complexifier l'analyse de sûreté des systèmes employant des plans d'exécutions SMP et hybrides. Néanmoins, le partitionnement robuste multicoeur demeure réalisable dans la pratique lorsque des précautions additionnelles sont prises, telles qu'une gestion adéquate des caches et l'emploi de plans d'exécution avec fenêtres d'activations critiques réservées sur un seul coeur. Des travaux additionnels sur la mitigation des interactions entre les coeurs et sur l'architecture des hiérarchies de mémoires permettraient de trouver des solutions aux problèmes les plus importants pour lesquels nous n'avons pas émis de pistes de solution. La réalisation de notre prototype XtratuM-PPC a d'ailleurs été motivée par l'absence de prototype permettant d'effectuer ces travaux sur une cible ou des modèles proches de la réalité.

Tableau 5.11 Résumé des problèmes trouvés
lors de la réalisation de XtratuM-PPC

Description	**Section**
Si une faute survenait lors de l'exécution de code dans la page des vecteurs d'exceptions déplacée, on ne peut plus se fier au contenu de cette page, puisqu'elle est en RAM. Le comportement du démarrage des autres coeurs serait donc imprévisible puisque la page est présumée comme étant corrompue. Une solution potentielle, mais non implémentée, serait l'utilisation de vecteurs de démarrage résidants en ROM.	5.3.4, p.137
Les registres contrôlant le réadressage ainsi que le démarrage des coeurs sont globalement accessibles par tous les coeurs. Il serait possible qu'une défaillance cause une écriture intempestive dans un de ces registres, ce qui pourrait engendrer le redémarrage inopiné d'un ou plusieurs coeurs. Nous n'avons trouvé aucune solution pouvant mitiger ce problème sur la plateforme MPC8641 évaluée.	5.3.4, p.138
Le redémarrage après une défaillance requiert un rendez-vous à une barrière globale et des partages de données entre les coeurs. La resynchronisation après le redémarrage d'un coeur est donc impossible, puisqu'il faudrait arrêter le traitement des autres coeurs pour traiter la resynchronisation qui est arrivée de manière asynchrone. Une alternative qui n'a pas été essayée est l'ajout d'une phase de resynchronisation d'horloge à la frontière de chaque bloc d'activation majeur.	5.4.4, p.141

Description	Section
La stabilité de la synchronisation des horloges locales, établie au démarrage du noyau, est strictement nécessaire pour garantir l'intégrité du partitionnement temporel et le respect du plan d'exécution. Toute désynchronisation, peu importe la raison, élimine les garanties d'isolation temporelles entre les partitions. Nous proposons l'ajout de ressources matérielles simples, dont une horloge locale à réinitialisation globale sur chaque coeur.	5.4.4, p.142
La structure de données d'émulation d'interruption, qui enregistre l'adresse de retour pour la restauration du contexte après une interruption, est enregistrée sur la pile de la partition active, ce qui rend le noyau à risque d'empoisonnement de la pile par une partition malicieuse. Une solution étudiée, mais non implémentée, serait l'utilisation d'une pile indépendante pour l'exécution de toutes les parties du noyau en mode superviseur.	5.5.3, p.148
L'instruction barrière `sync` sur les processeurs PowerPC multicoeurs émet des transactions d'ordonnancement vers le module de cohérence mémoire qui peuvent affecter le temps d'exécution sur les autres coeurs. Ces délais doivent être pris en considération par les outils d'analyse du pire cas de délai d'exécution.	5.6.4, p.158
Une boucle infinie d'instructions barrières peut affecter la performance des autres coeurs en saturant la file de transactions du module de cohérence mémoire avec des transactions de synchronisation, au détriment d'emplacements de file permettant des transactions utiles en mémoire. Une minuterie chien de garde pourrait borner la durée maximale de cette condition, mais nous n'avons pas implémenté cette solution.	5.6.4, p.158

Description	**Section**
Les variables partagées fréquemment doivent être alignées à la taille d'une ligne de cache dans le but d'éviter que la contention autour du partage ne cause d'invalidations superflues affectant des variables n'ayant aucun lien avec elles.	5.6.4, p.159
L'instruction de vidange d'entrées de TLB « tlbie » pourrait causer des invalidations d'entrées de TLB imprévues dans les autres coeurs si la diffusion des transactions globales de vidange est permise sur le bus de cohérence. Notre implémentation ne souffre pas de ce problème, car nous désactivons la diffusion des transactions globales de vidange lors du changement de contexte de partitions.	5.8.3, p.174
Les données du plan de configuration du système compilé en table de configuration binaire ne sont pas validées au démarrage. Il serait possible de corrompre volontairement la table de configuration pour forcer des configurations de MMU qui donnent des droits d'accès à des partitions qui brisent l'isolation spatiale. Une solution à ce problème serait d'employer des signatures de validation calculées à partir de la version de confiance de la table de configuration binaire.	5.8.3, p.175

CONCLUSION

À travers les chapitres qui ont précédé, nous avons démontré qu'il était possible d'adapter un noyau de partitionnement robuste monocoeur existant à une architecture multicoeur. De plus, nous avons documenté les considérations techniques et les problèmes potentiels liés à l'exploitation des processeurs multicoeurs par un noyau de partitionnement robuste.

Le modèle de partitionnement robuste multicoeur réalisé atteint les objectifs initiaux :

- il permet la consolidation d'un plus grand nombre d'applications avioniques sur une même machine en exploitant des processeurs multicoeurs existants ;
- il offre aux chercheurs la possibilité d'explorer les principaux modèles de partitionnement temporels parallèles (AMP, SMP et hybride) ;
- il permet le prototypage d'applications IMA dans un environnement réel, comme les solutions déjà existantes dans le domaine commercial.

Les problèmes que nous avons relevés sont résumés aux sections 4.2 et 5.11. Nous avons proposé une contextualisation de ces problèmes qui saura être utile autant aux autres chercheurs qu'aux praticiens dans notre domaine.

En rétrospective, le concept du partitionnement robuste multicoeur n'est pas incompatible avec la sûreté logicielle. D'ailleurs, l'exploitation de plans d'exécution multicoeur est déjà permise par le système d'exploitation avionique PikeOS du domaine commercial. Les obstacles réels sont plutôt le manque de prédictibilité du temps d'exécution sur les processeurs PowerPC multicoeurs actuels et l'apparition de nouveaux besoins de synchronisation logicielle qui augmentent la complexité du code d'application dans les partitions.

Le développement d'une architecture multicoeur compatible avec les besoins de l'évaluation du pire cas de temps d'exécution augmenterait la confiance de l'industrie envers les noyaux de

partitionnement robuste multicoeur. Les travaux du projet MERASA [64] ont fait progresser l'état de l'art à ce sujet.

En ce qui a trait aux besoins de synchronisation additionnels, cette catégorie de problème peut être mitigée en n'exploitant que des plans d'exécution AMP qui ne requièrent aucune synchronisation entre les coeurs. Ce type de plan d'exécution ne permet pas d'exploiter le parallélisme au niveau des tâches sur plusieurs coeurs pour augmenter les performances d'applications individuelles existantes. Cependant, les plans d'exécution AMP permettent d'augmenter globalement la quantité de fonctions intégrées sur une plateforme IMA, ce qui est tout de même un avantage en comparaison avec l'approche monocoeur actuelle.

Malgré l'atteinte de nos objectifs, nous sommes conscients des limitations de notre prototype. En particulier, XtratuM-PPC ne peut être utilisé qu'à des fins d'évaluation et de recherche. Nous avons basé notre implémentation sur l'état de l'art et développé un modèle qui nous semblait intuitif et cohérent, en considérant des cas d'utilisation préssentis et des exigences réelles. Cependant, il était impossible d'implémenter tous les aspects d'un système d'exploitation avionique commercial, en raison du cadre d'un projet de maîtrise. En conséquence, il est fort possible que certaines exigences industrielles essentielles aient été complètement omises, puisque nous n'avions pas accès à une équipe de conception de systèmes IMA pour valider nos exigences. Similairement, le niveau d'assurance-qualité appliqué aux livrables ne permet pas de garantir la sûreté de notre noyau avec une certitude suffisante pour le déploiement dans des vrais appareils. Finalement, certains obstacles que nous avons relevés par rapport à la réalisation d'un noyau de partitionnement robuste multicoeur demeurent non résolus.

Outre ces limitations, le prototype de noyau de partitionnement robuste multicoeur que nous avons contribué fournit une base de comparaison entre les technologies actuelles et les solutions qui pourraient être développées dans le futur.

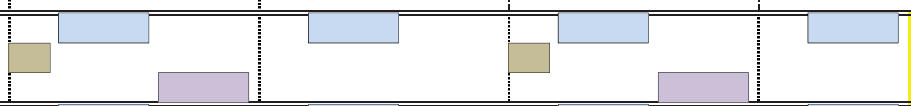
La genèse de nos travaux se trouvait dans l'absence de solutions et résultats de recherche dans le domaine des systèmes avioniques employant les processeurs multicoeurs. D'ailleurs, le domaine de l'IMA sur processeurs monocoeurs est encore en effervescence et le modèle IMA est loin d'être déployé dans tous les appareils en vol. Malgré tout, puisque les processeurs multicoeurs sont déjà la norme en informatique générale et que la complexité des systèmes avioniques est croissante, nous croyons qu'il était important de démarrer les recherches pour ouvrir la porte au déploiement inévitable de ces processeurs dans les avions commerciaux. Les problèmes que nous avons identifiés ainsi que le prototype de noyau que nous avons réalisé seront utiles aux autres chercheurs qui désirent faire progresser l'état d'avancement de l'utilisation des processeurs multicoeurs en avionique.

ANNEXE I

TABLEAUX ET FIGURES SUPPLÉMENTAIRES

1 Plan d'exécution de l'étude de cas

Se référer au tableau inséré à la page suivante.

CPU	ID	Application	0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95	Période	%CPU utilisé
0	0	FFT		50	36
0	3	IVHM		100	8
0	4	IO		50	18
1	1	FFT		50	36
1	5	LZ		25	16
2	2	FFT		50	36
2	6	LZ		25	16

Inoccupé

CPU0		
Application	Décalage (ms)	Durée (ms)
IVHM	0	4
FFT	5	9
IO	15	9
FFT	30	9
IVHM	50	4
FFT	55	9
IO	65	9
FFT	80	9

CPU1 et CPU2		
Partition	Décalage (ms)	Durée (ms)
LZ	0	4
FFT	5	9
LZ	25	4
FFT	30	9
LZ	50	4
FFT	55	9
LZ	75	4
FFT	80	9

2 Copie de la console du système de l'étude de cas

```
CPU:
    Core: E600, Version: 0.0, (0x80040000)
    System: 8641D, Version: 0.0, (0x80900100)
    Clocks: CPU: 792 MHz, MPX: 264 MHz, DDR: 132 MHz,LBC:
       unknown (lcrr: 0x00000000)
    L2: Enabled
    ASMP: Off
Board: Virtutech Simics MPC8641-simple
I2C: ready
DRAM: No memory modules found for DDR controller 2!!
Non-interleaved DDR: 512 MB
FLASH: 16 MB
*** Warning - bad CRC, using default environment
Read failed.
PCI-EXPRESS 1: Disabled
PCI-EXPRESS 2 on bus 00 - 00
No radeon video card found!
In: serial
Out: serial
Err: serial
Net: eTSEC1, eTSEC2, eTSEC3, eTSEC4
Hit any key to stop autoboot: 0
=> go 40000 -debug
## Starting application at 0x00040000 ...
XM-PPC Hypervisor (2.2.2)
>> OpenPIC rev1.2 found
Number of CPUs supported on SoC: 3
Number of IRQs supported on SoC: 84
MPC86xx FRR[NIRQ] errata present: number of IRQ sources
   adjusted.
Detected cpu0 clocked at 792.0MHz
[PMM] Physical memory region [0x0-0x20000000] type: 1
>> HwTimers: (PowerPC OpenPIC Timer A2, PowerPC OpenPIC Timer
   A3, PowerPC Decrementer Timer (per-cpu))
>> HWClocks: (PowerPC Timebase clock (per-cpu))
System clock: PowerPC Timebase clock (per-cpu) [66000Khz]
[CPU:0] [hwTimer] using PowerPC Decrementer Timer (per-cpu)
   [66000Khz]
[CPU:1] [hwTimer] using PowerPC Decrementer Timer (per-cpu)
   [66000Khz]
[CPU:2] [hwTimer] using PowerPC Decrementer Timer (per-cpu)
   [66000Khz]
[CPU:0] [sched] using cyclic scheduler
7 Partition(s) created
P0 ("FFT0":0) cpu: 0 flags: [ BOOT (0x1000000) SMP MASTER ]:
```

```
    "FFT_image" [0x1000000 - 0x1161000]
[CPU:2] [sched] using cyclic scheduler
P1 ("FFT1":1) cpu: 1 flags: [ BOOT (0x1000000) SMP ]:
    "FFT_image" [0x1000000 - 0x1161000]
P2 ("FFT2":2) cpu: 2 flags: [ BOOT (0x1000000) SMP ]:
    "FFT_image" [0x1000000 - 0x1161000]
P3 ("IVHM":3) cpu: 0 flags: [ SV BOOT (0x1180000) MASTER ]:
    "IVHM_image" [0x1180000 - 0x11c9000]
P4 ("IO":4) cpu: 0 flags: [ BOOT (0x1200000) MASTER ]:
    "IO_image" [0x1200000 - 0x12a9000]
    "IO_xdata" [0x1400000 - 0x1600000]
P5 ("LZ1":5) cpu: 1 flags: [ BOOT (0x1300000) SMP MASTER ]:
    "LZ_image" [0x1300000 - 0x13b1000]
P6 ("LZ2":6) cpu: 2 flags: [ BOOT (0x1300000) SMP ]:
    "LZ_image" [0x1300000 - 0x13b1000]
[CPU:0] Joined at BarrierWait(&g_smpPartitionInitBarrier)
[CPU:1] Joined at BarrierWait(&g_smpPartitionInitBarrier)
[CPU:2] Joined at BarrierWait(&g_smpPartitionInitBarrier)
[CPU:1] Mean: 12683, Variance: 1387
[CPU:2] Mean: 12655, Variance: 1390
[CPU:0]
CPU 0 time: 124243
CPU 1 time: 136954
CPU 2 time: 136931
----------
```

3 Extrait du journal d'exécution de l'étude de cas

```
[cpu1:XM] (182156 us) Switching from [Idle1@0x41fd58] -> [
   FFT1@0x42d380] from irq_return_restore_context->
   DoPpcDecrementer->Scheduling
### TIMING CONTEXT_SWITCH_OUT@0x0041fd58 197221924 197221992
   68
### TIMING SLOT@0x0041fd78 196430460 197221886 791426
[cpu2:XM] (182157 us) Switching from [Idle2@0x41fd78] -> [
   FFT2@0x42d3a0] from irq_return_restore_context->
   DoPpcDecrementer->Scheduling
### TIMING CONTEXT_SWITCH_OUT@0x0041fd78 197221888 197221956
   68
### TIMING SLOT@0x0041fd38 192470638 197222064 4751426
[cpu0:XM] (182156 us) Switching from [Idle0@0x41fd38] -> [
   FFT0@0x42d360] from irq_return_restore_context->
   DoPpcDecrementer->Scheduling
### TIMING CONTEXT_SWITCH_OUT@0x0041fd38 197222066 197222134
   68
### TIMING MMU@0x0042d380 197222001 197222881 880
### TIMING MMU@0x0042d3a0 197221965 197222845 880
### TIMING MMU@0x0042d360 197222143 197223023 880
### TIMING CONTEXT_SWITCH_IN@0x0042d360 197222136 197223233
   1097
### TIMING CONTEXT_SWITCH_IN@0x0042d380 197221994 197223091
   1097
### TIMING CONTEXT_SWITCH_IN@0x0042d3a0 197221958 197223055
   1097
[FFT0] Received and enqueued from I/O !
[FFT1] dequeued a received message
[FFT0] Received and enqueued from I/O !
[FFT2] dequeued a received message
[FFT0] Received and enqueued from I/O !
[FFT0] Received and enqueued from I/O !
[FFT0] Received and enqueued from I/O !
[FFT0] Received and enqueued from I/O !
### TIMING FFT_PROCESS@0x0042d380 197279629 198441455 1161826
[FFT1] enqueued a message for IO
[FFT1] dequeued a received message
### TIMING FFT_PROCESS@0x0042d3a0 197328630 198490456 1161826
[FFT0] Received and enqueued from I/O !
[FFT2] enqueued a message for IO
[FFT2] dequeued a received message
```

ANNEXE II

ARTICLE DE CONFÉRENCE SUR LA MÉTHODE VPI

Les sept pages suivantes reproduisent l'article que nous avons soumis à la conférence « IEEE International Symposium on Rapid System Prototyping » de 2011.

La référence bibliographique de l'article est la suivante :

> Carmel-Veilleux, Tennessee, Jean François Boland et Guy Bois : *A Novel Low-Overhead Flexible Instrumentation Framework for Virtual Platforms.* Dans *Proc. 22nd IEEE International Symposium on Rapid System Prototyping*, mai 2011.

L'article est reproduit dans les sept pages suivantes.

A Novel Low-Overhead Flexible Instrumentation Framework for Virtual Platforms

Tennessee Carmel-Veilleux*, Jean-François Boland* and Guy Bois†

* Dept. of Electrical Engineering, École de Technologie Supérieure, Montréal, Québec, Canada
† Dept. of Software and Computer Engineering, École Polytechnique de Montréal, Montréal, Québec, Canada

***Abstract*—Instrumentation methods for code profiling, tracing and semihosting on virtual platforms (VP) and instruction-set simulators (ISS) rely on function call and system call interception. To reduce instrumentation overhead that can affect program behavior and timing, we propose a novel low-overhead flexible instrumentation framework called Virtual Platform Instrumentation (VPI). The VPI framework uses a new table-based parameter-passing method that reduces the runtime overhead of instrumentation to only that of the interception. Furthermore, it provides a high-level interface to extend the functionality of any VP or ISS with debugging support, without changes to their source code. Our framework unifies the implementation of tracing, profiling and semihosting use cases, while at the same time reducing detrimental runtime overhead on the target as much as 90 % compared to widely deployed traditional methods, without significant simulation time penalty.**

***Index Terms*—Computer simulation, Software debugging, Software prototyping, System-level design**

I. Introduction

With the advent of multiprocessor systems-on-chip (MPSoC) for consumer and networking applications, complexity has become a significant issue for system debugging and prototyping. Simulators and system-level modeling tools have become necessary tools to manage this complexity. Virtual platforms (VP) are system-level software tools combining instruction-set simulators (ISS) and peripheral models that are used to start software prototyping before availability of the final product. In the case of state-of-the-art MPSoCs, virtual platform models can even be used as the "golden model" provided to developers years before availability of final silicon [1]. The proliferation of SystemC-based design-space exploration tools (e.g. Platform Architect [2], ReSP [3], Space Studio [4], etc.) was also made possible by mature VP technology.

When using VPs for debugging or design-space exploration, software instrumentation methods can be used to obtain profiling data, execution traces or other introspective behavior. The runtime overhead (i.e. *intrusiveness*) on the target of these instrumentation methods is critical. It must be minimized to prevent interfering with the strict timing constraints common in embedded software [5].

In this paper, we present a novel low-overhead flexible code instrumentation framework called *Virtual Platform Instrumentation* (VPI). The VPI framework can be used to extend existing virtual platforms with additional tracing, profiling and semihosting capabilities with minimal target code overhead and timing interference. Semihosting is a mechanism whereby a function's execution on the target is delegated to an external hosted environment, such as a VP. Semihosting is traditionally used to exploit the host's I/O, console and file system before support becomes available on the target [6].

Through our proposed framework, we make three main contributions.

Firstly, we describe a new mechanism for fully inlinable instrumentation insertion with table-driven parameter-passing between a simulated target and its host. Our method completely foregoes function call parameter preparation overhead seen in traditional semihosting. In doing so, we reduce detrimental runtime overhead on the target between 2–10 times in comparison to traditional methods, while showing nearly identical simulation run times.

Secondly, we show that our framework can realize function semihosting, tracing and profiling tasks, thus unifying usually separate use cases.

Thirdly, we propose a generic high level instrumentation handling interface for VPs which allows for new instrumentation behavior to be added to existing tools, without requiring modifications.

This paper is organized as follows: section II presents background information and related works about virtual platform code instrumentation, section III describes our proposed instrumentation framework, section IV presents experimental case studies of semihosting and profiling with conclusions and future work in section V.

II. Background and related work

In this section we explore different instrumentation methods used for debugging, system prototyping and profiling on virtual platforms. This is followed by an overall comparison of the methods, including our proposed VPI framework.

For our purposes, we define *virtual platforms* as software environments that simulate a full *target system* on a *host platform*. Virtual platforms integrate *instruction-set simulators* as well as models of memories, system buses and peripherals to realize a full SoC simulator. The conceptual layering of a VP is shown in Figure 1. Through the development of our framework, we evaluated the features and mechanisms present in the Simics [7], Platform Architect [2], QEMU [8], ReSP [3] and OVPSim [9] virtual platforms. In our experimental case study of section IV, we concentrated on Simics and QEMU.

A. Instrumentation use cases overview

We define *instrumentation* as tools added to a program to aid in testing, debugging or measurements at run-time. These tools can be implemented as *intrusive* instrumentation functions in source code or as *non-intrusive* instrumentation functionality within a VP. In our context, intrusive means that target run time is affected in some way by the instrumentation. An *instrumentation site* refers to the location where instrumentation is inserted.

The authors would like to acknowledge financial support from the Fonds québécois de la recherche sur la nature et les technologies (FQRNT), the École de technologie supérieure (ÉTS) and the Regroupement Stratégique en Microsystème du Québec (ReSMiQ) in the realization of this research work.

978-1-4577-0660-8/11$26.00 © 2011 IEEE

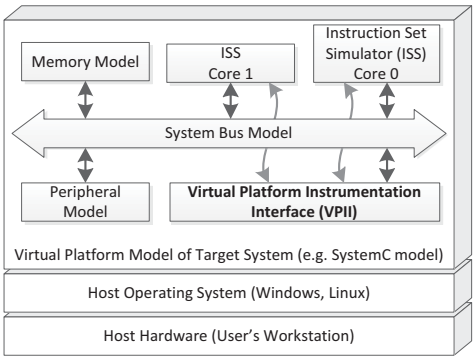


Figure 1. Virtual platform modeling layers

Some examples of intrusive instrumentation use cases are:

- compile-time insertion of tracing or profiling calls at every function entry and exit point [10, p. 75];
- compile-time insertion of code coverage or other measurement statements in existing source code;
- insertion of probe points for fine-grained execution tracing at the OS kernel level (e.g. Kernel Markers [11] in the Linux kernel).

Conversely, examples of non-intrusive instrumentation include:

- insertion of breakpoints and watchpoints at runtime using a debugger to aid in tracing and debugging;
- interception of library function calls through their runtime address to emulate functionality or store profiling data [4], [3];
- runtime insertion of transparent user-defined instruments tied to program or data accesses such as the probe, event and watch mechanisms of Avrova [12];
- storage of control-flow data in hardware trace buffers readable through specialized interfaces.

The instrumentation methods which implement these use cases differ significantly in how much they affect target and host run time (i.e. their intrusiveness) and what they enable the user to do (i.e. their flexibility).

In terms of tracing instrumentation, many varieties exist which differ in semantic level. The VPs we evaluated each allowed for straightforward dumping of a trace showing *all instructions* executed and *every data access*, with no context-related semantic information. Conversely, user-defined or compiler-inserted high-level tracing stores much less data, but with much higher semantic contents (e.g. a list of task context switches in an OS [7]). When we refer to tracing in this paper, we are referring to the high-level tracing case.

In the next section we will discuss *semihosting*, a method used to implement several of the aforementioned instrumentation tasks either intrusively or non-intrusively.

B. Semihosting function calls

In the general case, semihosting works by intercepting calls to specific function stubs in the target code. Instead of running the function on the target at these sites, the ISS forwards an event to the VP. The VP's semihosting implementation then examines the processor state in the ISS and emulates the function's behavior appropriately, with target time stopped.

With semihosting, mechanisms for *call interception* differ by implementation. Run-time and code size intrusiveness are directly linked to which interception mechanism is used. We distinguish three such mechanisms:

- *"Syscall" interception:* the "system call" instruction is diverted for use in interception. This is the traditional approach as used by ARM tools [6] as well as in QEMU for PPC and ARM platforms. This approach may require an exception to be taken, with associated runtime overhead.
- *"Simcall" interception:* a specific instruction is diverted for use as an interception point. The instruction can be specific for that purpose, like the SIMCALL instruction in the Tensilica Xtensa architecture [13, p. 520], or it can be an architectural NO-OP as in the Simics virtual platform, where it is called a "magic instruction" [7].
- *Address interception:* the entrypoints of all functions to be emulated are registered as implicit breakpoints in the VP. When the program counter (PC) reaches these breakpoints, interception occurs. This approach is used in tools such as Imperas OVP-Sim [9], ReSP [3] and Space Studio [4] amongst others. It can also be implemented in any VP with debugging support using watchpoints or breakpoints.

With all traditional semihosting methods, context-specific parameters are passed using regular function parameter-passing. The preparation of semihosted function call parameters according to normal calling conventions accounts for most of the runtime overhead of these methods.

Since the emulated function is fully executed by VP host code, the entire state of the modeled system can be exploited. Using the example of a MPSoC model, this would imply that internal registers of all CPU cores could be accessed while processing the emulated function. This opens the possibility of emulating as much as a full OS system call as done in [3], [14] or providing "perfect" barrier synchronization primitives across the system [15].

Semihosting is increasingly used in the implementation of design space exploration tools for hardware–software codesign. In that case, it enables developers to quickly assess the performance of an algorithm without having to deal with the accessory details of OS porting or adaptation early in the design phase [3], [4].

C. Comparison of instrumentation methods

In this section we compare different instrumentation methods commonly used under virtual platforms. The comparison matrix is shown in Table I. Although we have not yet detailed our Virtual Platform Instrumentation (VPI) method, it is included in the Table for comparison purposes and fully described in section III.

Firstly, we evaluated intrusiveness in terms of code size, run time and features "lost" to the method (e.g. system call no longer available). Secondly, we established whether the methods work without symbol information (i.e. even with a raw binary image) and whether they allow for inlining of the instrumentation. By symbol information, we mean the symbols table that links function names to their addresses, which is present in all object file formats. Finally, we determined if the methods listed were suitable for different use cases presented earlier. For the qualitative criteria, we evaluated implementation source code or manuals of every method listed to determine the values shown.

Our VPI method appears to compare favorably with existing approaches. We contrast our method with other approaches and provide experimental results supporting these intrusiveness comparisons in section IV.

Table I
INSTRUMENTATION METHODS COMPARISON MATRIX

Method	Intrusive ?			Works ?		Supports ?		
	Code	Run-time	Features lost	Without symbols	Inline	Tracing	Profiling	Syscall/OS emulation
Compiler-inserted profiling function calls	High	High	No	No	No	Yes	Yes	No
Traditional semihosting ("Syscall" interception)	Medium	Medium	Yes	Yes	No	Depends	Depends	Yes
Traditional semihosting ("Simcall" interception)	Low	Low	Yes	Yes	No	Depends	Depends	Yes
Traditional semihosting (Address interception)	Low	None	No	No	No	Yes	Yes	Yes
Watchpoints / Breakpoints	None	None	No	No	N/A	Yes	No	No
VPI (Proposed method)	**Low**	**None–Low**	**No**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

III. DETAILS OF PROPOSED FRAMEWORK

Our code instrumentation framework (VPI) is composed of two software elements:

1) an inline instrumentation insertion method with table-based parameter-passing, implemented with inline assembler in C code;
2) a high-level *virtual platform instrumentation interface* (VPII) that handles interception of instrumentation sites by calling appropriate *virtual platform instrumentation functions* (VPIFs).

Combined, these two components form a low-overhead generic code instrumentation framework that can be implemented on any VP or ISS with debugger support or extension capabilities.

For the purposes of this paper, the compiler's inline assembler extensions are those of the unmodified GCC version 4.5 C compiler [16]. However, the concepts behind our method are tool-agnostic and applicable to production-level compilers.

In the following subsections, we refer to the numbered markers in Figure 2 to illustrate the flow of instrumentation insertion from initial source code to the compiler-generated assembler code. Marker 1 of Figure 2 will be listed as (❶), marker 2 as (❷) and so on. We will use the `fopen()` C library function as a semihosting example to illustrate instrumentation insertion.

A. Target-side instrumentation insertion

Instrumentation statements are inserted into target code by the developer using common C macros (❶). They can refer to any program variable (❷). Each instrumentation macro expands to inline assembler statements containing a semihosting *interception block* (❸,❹) and a *parameter-passing payload* related to the desired instrumentation (❺). The entire instrumentation call site is inserted inline (i.e. in-situ).

At compile time, the interception block (❼) and parameter-passing payload table (❽) are constructed from compiler-provided register and memory address allocations. This is done by accessing inline-assembler-specific placeholders (❻) and pretending instructions are emitted from them.

When inline assembler is used within a function's body, placeholders referencing C variables in the assembler code are replaced by values from the compiler's internal registers and memory addresses allocation algorithms.

We save these references out of band from the main code section (".text"), in the read-only data section (".rodata"). The choice of the ".rodata" section for the payload data table is deliberate, to prevent instruction cache interference by data that never gets read by user code. However, it is possible and sometimes required to use the ".text" section for the payload table. For example, if the target OS uses paged virtual memory, the interception block and payload table may need to be inlined in the code section. Otherwise, the table's effective address range might not currently be mapped-in by the OS, causing a data access exception at interception time.

Interception block

Although interception is still necessary with our method, we do not mandate the use of a specific mechanism. The interception block from our example of Figure 2 (❸,❹) is composed of three parts: 1) "Simcall" interception instruction ("`rlwimi 0,0,0,0,9`" in this case); 2) pointer-skipping branch; and 3) payload table pointer. The interception block shown is an arbitrary example. Any other interception mechanism described in section II-B could be used, as long as it is supported by the VP.

Along with the interception block, a pointer to the parameter-passing payload table is used to link an instrumentation site with its parameters. An unconditional branch is added to the interception block to prevent the fetching and execution of the payload table pointer.

Parameter-passing payload table

The parameter-passing payload table serves as a link between the target program's state and the high-level instrumentation interface running in the VP. For an instrumentation site, it both uniquely identifies desired behavior and provides reference descriptors to the function parameters that should be passed to–from the handler. These reference descriptors allow a high-level instrumentation interface to both read data from, and write data back to the target program's state.

The format used for each payload table is as follows:

- Signature header (1 word) including a functional identifier (16 bits) and quantity (from 0–15 each) of constants, input variables and output variables references;
- Constants table (1 word each);
- Input variables references (fixed number of strings and/or instructions);
- Output variables references (fixed number of strings and/or instructions);

The *signature header* identifies the desired functional behavior (e.g.: tracing, `fopen()`, `printf()`, etc.). For every functional identifier, it is possible to use more or less constants, inputs and outputs depending on the need. For instance, a "`printf()`" function could be implemented as 16 versions, covering the cases where 0 to 15 variables need to be formatted.

Constants are emitted from references known before runtime. For instance, our implementation of a semihosted "`printf()`" uses a

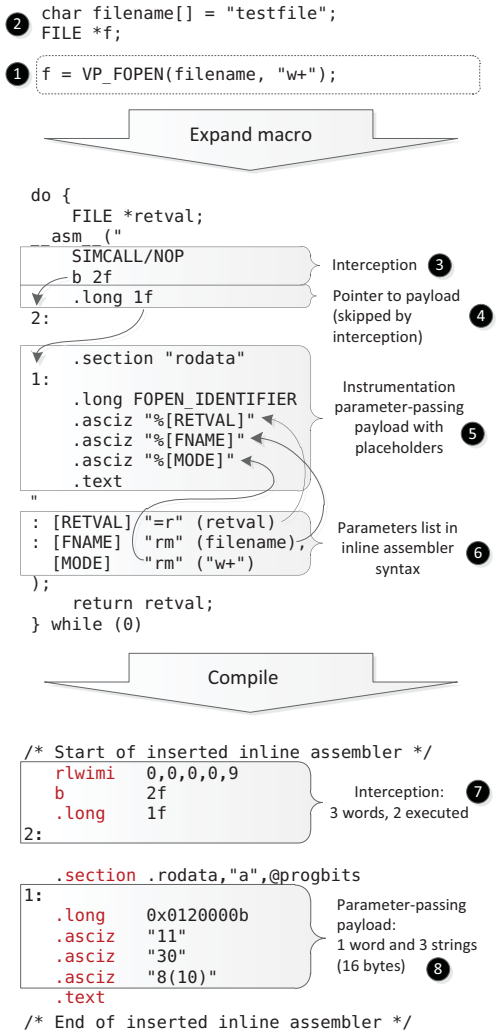


Figure 2. Overview of VPI instrumentation insertion in source code

constant slot for the pointer to the format string. The example of Figure 2 does not use any constants.

Input variable references and *output variable references* are compiler-provided data references that can be accessed by the VPIFs through the VPII.

Table II breaks-down the payload table of our "`fopen()`" example from Figure 2. Again, this example is based on a PowerPC target, but equivalent content would be present for any architecture.

Although our example of Figure 2 uses only strings for references, both strings and instructions can be used, as long as the table format is understood by the VPII implementation. In the case where instructions are used, the VPII can disassemble them at runtime to decode the references they contain. To illustrate this, we show a store

Table II
DESCRIPTION OF FIGURE 2'S PAYLOAD TABLE

Compiled value	Description
0x0120000b	Signature header • Function 0x000b • 0 constants • 2 inputs • 1 output
"11"	Value of "retval" output variable is in GPR11
"30"	Value of "filename" input variable is in GPR30
"8(10)" or `stw 0,8(10)`	Pointer to "mode" ("w+") input variable is contents of GPR10 + 8

("`stw`") instruction that could replace the reference string of the last reference in the Table.

Remarks about insertion method construction

As far as we know, every other semihosting-based methods are designed for *source code equivalence*—all instrumentation-calling code must remain identical after instrumentation is removed. This requirement has the advantage of allowing instrumentation to be included by simply linking with different versions of the libraries. However, parameter-passing becomes bound to the C calling conventions in effect on the target platform. We constructed our proposed instrumentation insertion method to overcome the artificial requirement of function call setup when running on virtual platforms.

In our case, where we know the instrumented binary will be run in a VP, it becomes only necessary to *somehow* tell the VP *where* to find function parameters after a call is intercepted. Function call preparation merely copies program variables into predetermined registers or stack frame locations. Since the VP can access all system state "in the background" without incurring instruction execution penalties, we replaced the function call and associated execution overhead with a static parameter-passing table. Parameters can then be accessed by interpreting the table, rather than reading predetermined registers or stack frame locations. The compiler guarantees the "reloading" from memory of any variable not locally available from registers or offsetable memory locations. This reloading overhead is, in all cases, a subset of standard function call overhead.

Another side effect of our method's construction is that instrumentation insertion is always inlined. This has the desirable consequence of "following" other inlining done by the compiler. It then becomes trivial to instrument functions inlined by the compiler's optimizer, and to identify them uniquely, without any special compiler support.

Finally, while optimizing compilers can reorder statements around sequence points in C code, some compiler-specific mechanisms can be used to guarantee the positioning of the inlined assembler blocks. During our tests with GCC 4.5 on ARM and PPC platforms, the use of `volatile asm` statements with "`memory`" clobber prevented any instruction reordering from affecting the test result signatures at every optimization level.

B. Virtual Platform Instrumentation Interface

Within our framework, we propose that the VP be pre-configured to run a centralized instrumentation handler whenever interception occurs at an instrumentation point. A high-level, object-oriented *virtual platform instrumentation interface* (VPII) layer is used to interface between the VP and the instrumentation functions by providing abstract interfaces to the VP's state and parameter-passing tables.

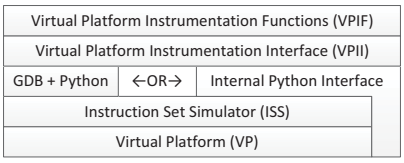


Figure 3. VPI framework implementation layers

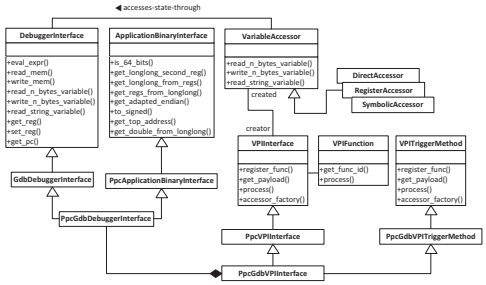


Figure 4. Class hierarchy of a sample VPII implementation

It also executes the appropriate *virtual platform instrumentation function* (VPIF) handler on behalf of the target code. The layers forming this high-level interface are shown in Figure 3.

The VPII abstraction allows the VPIF handlers to access registers, memory and internal VP state using generic accessors that hide low-level platform interfaces. It also handles data conversion tasks related to a platform's application binary interface (ABI).

The VPII is implemented using the high-level language (HLL) extension interfaces built into VPs. For instance, this could be an internal script interpreter, such as Python in Simics [7] or Tcl in Synopsis Platform Architect tools [2]. It could also be a C++ library built on top of a SystemC simulator. Alternatively, the GNU Debugger's (GDB) Python interface can be used to implement a generic VPII suitable for existing ISS and VP implementations with GDB debugging support.

For our experimental implementation, we developed VPII and VPIF libraries supporting both Simics' and GDB's Python extension interfaces. The class hierarchy for our implementation of the VPII interface for PowerPC targets with GDB-based VP access is shown in Figure 4. In that example, the `PpcGdbVPIInterface` class is used as the focal point to register instrumentation behavior (VPIF handlers) and access the VP through GDB.

For testing, we also developed a sample library of VPIF handlers covering common instrumentation tasks of I/O semihosting, tracing and code timing. New VPIF handlers can be registered and modified dynamically at run-time with our sample Python implementation. Handlers written generically using only parameter accessors without using any VP-specific functionality can be reused on any supported architecture.

IV. Experimental case study

In order to validate that the method we propose is flexible and has low overhead, we performed a comparative case study. We compared our experimental framework implementation to common instrumentation methods using controlled examples.

Source code for the case study, as well as for our VPI implementation, is available at http://tentech.ca/vp-instrumentation/ under a BSD open-source license.

A. Experimental setup

The case study was run on a standard PC running Windows 7 x64 Professional with an Intel Core 2 Duo P8400 with two 2.26GHz cores. The toolchain and C libraries were from the Sourcery G++ 2010.09-53 release, based on GNU GCC 4.5.1 and GNU Binutils 2.20.51. The target was a PowerPC e600 single-core processor on the Wind River Simics 4.0.60 and QEMU PPC 0.11.50 virtual platforms. We used GDB 7.2.50 with Python support as the debugger.

We instrumented the "QURT" quadratic equation root-finding benchmark program from the SNU WCET suite [17] based on two instrumentation scenarios, which were run independently. These scenarios showcase the unification of profiling and semihosting use cases since both are implemented using the same VPI framework functionality and insertion syntax.

Each scenario comprised a base non-instrumented case, and four instrumented cases. The instrumented cases represent different combinations of instrumentation methods and VPI configurations. The VPI configurations were the following:

- Internal: VPI handler is run internally on Simics' Python interpreter with "simcall" interception.
- External: VPI handler is run externally on GDB's Python interpreter with debugger watchpoint interception under either Simics or QEMU.

We compared the three following instrumentation methods:

- "VPI": uses our inlined VPI instrumentation for each site;
- "Stub-call": calls a C function stub at every site which wraps an inlined VPI instrumentation site, so that traditional semihosting function call overhead can be compared;
- "Full-code": in the case of the `printf()` scenario, we run an optimized `printf()` implementation entirely on the target, with I/O redirected to a null device so that "manual" non-semihosted instrumentation overhead can be compared.

For each run, we recorded binary section sizes, simulation times on the host and cycle counts on the target. Section sizes provide information about code size interference. Simulation times and cycle counts are used to compare runtime overhead. With the "stub-call" cases, the results in Tables III, IV and V are compensated by subtracting the wrapped VPI site contribution, which would have artificially inflated the results of those cases.

All results are from release-type builds with no debugging symbols and "`-O2`" ("optimize more") option on GCC. Host OS noise was quantified by executing 50 runs of each case.

B. Results of "printf()" semihosting scenario

The *printf() semihosting scenario* compares space and time overhead of a `printf()` function semihosting use case. In this case, we inserted 3 instrumented sites to display the results of different loops of the QURT benchmark. Each loop ran 100 times, for a total of 300 calls. For all cases, the `printf()` implementation was functionally equivalent, with full float support. The `printf()` statement was `printf("Roots: x1 = (%.6f%+.6fj) x2 = (%.6f%+.6fj)\n", x1[0], x1[1], x2[0], x2[1])`.

Table III
TARGET RUNTIME OVERHEAD IN CYCLES FOR `PRINTF()` SCENARIO

Instrumentation case	CPU cycles	Total overhead	Per-call overhead	Overhead increase
None	**5 538 648**	**0**	**0**	N/A
Internal VPI	5 539 272	624±24	2	×1 (Base)
External VPI	5 539 872	1224±24	4	×2
Stub-call	5 545 848	6888±24	23	×11.5
Full-code	7 174 476	1 635 828±24	5453	×2726.5

Table IV
BINARY SIZE OVERHEAD FOR `PRINTF()` SCENARIO

Instrumentation case	`.text` size	`.data` size	`.rodata` size	Total
None	**34 796**	**1864**	**1224**	**37 884**
Internal VPI	+132	+0	+200	+332
External VPI	+168	+32	+200	+400
Stub-call	+320	+0	+80	+400
Full-code	+328	+0	+40	+368

We ran this scenario on both Simics and QEMU. QEMU only supports the external configuration without source code modifications. Since all binaries are identical between Simics and QEMU, the results of Tables III and IV apply equally to both VPs.

Simulated CPU runtime overhead results are detailed in Table III. Uncertainty on overhead was ±24 cycles because of the timing method. We observe that execution overhead per call for VPI cases is only 2–4 cycles, depending on the configuration. The external VPI configuration—with watchpoint interception—requires twice as many instructions per call as the internal "simcall"-based VPI configuration. Overheads of the VPI cases are a significant 5–11 times reduction over traditional stub-call instrumentation. Function call preparation accounts for the higher overhead of the stub-call case. In contrast, even when excluding I/O cost, the full-code `printf()` cases has 3 orders of magnitude higher runtime overhead than either VPI cases.

Space overhead results are listed in Table IV, in comparison to the uninstrumented base case. Code section (.text) space overheads of the VPI cases are noticeably lower than the other cases. Through manual assembler code analysis we confirmed that function call preparation accounted for the difference observed between the stub-call and full-code cases. As expected, the lower code section overheads with VPI cases come at the cost of a larger constants section (.rodata), although total sizes are comparable.

In terms of simulation time, our VPI framework's overhead depends considerably on whether an internal or external configuration is used. Simulation times for different scenarios under both Simics and QEMU are shown in Figure 5 (note the logarithmic scale on the simulation time axis). The "full -code" and "stub-call" cases in that figure do not have any interception methods enabled at runtime.

The internal uninstrumented ("Internal None") case is shown to have no penalty on simulation time. Conversely, the external uninstrumented ("External None") case—which uses watchpoint instead of "simcall" interception—causes some baseline interception overhead. Furthermore, the internal VPI-only case is shown to have *no penalty* over a traditional internal stub-call case.

With all instrumented cases, those using internal configurations

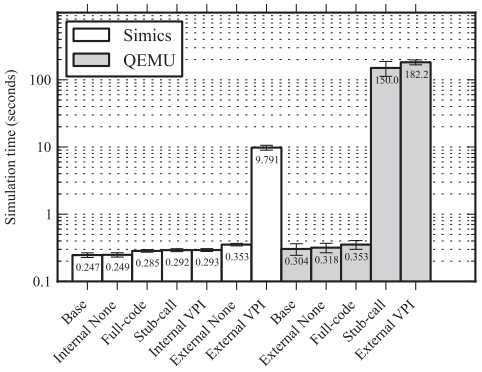


Figure 5. Simulation times for `printf()` scenario

display significantly better simulation performances than those using external configurations. Moreover, the internal VPI instrumented case is even faster than the external uninstrumented case under Simics. This shows that simulation time is practically unaffected by low instrumentation loads when an internal VPI framework configuration is used. The interception and VPII mechanisms appear to be much slower when going through the GDB interfaces used in all external cases. We determined that the slowdown was due to the overhead of both the GDB ASCII protocol and the context switches required to go back and forth between the GDB and VP processes. In contrast, the internal configuration has direct access to VP resources, which explains its better performances. In the case of QEMU, GDB communication overhead was prohibitive enough to prevent the use of our framework for non-trivial cases under that particular VP.

C. Results of "profiling" scenario

The *profiling scenario* compares overhead of runtime profiling between stub-call (i.e. compiler-inserted) and inlined tracing/profiling instrumentation. In the stub-call cases, the `-finstrument-functions` option of GCC was used to automatically insert a call to instrumentation stubs at every function entry and exit points. For the VPI cases, we manually inserted the VPI tracing calls in the C source code at every function entry and exit points. In both cases, the instrumentation behavior involved recording execution tracing information to a file, as usually done by profiling tools. The tracing call was of the form `vp_gcc_inst_trace("FUNC_ENTER", "NAME", __FILE__, __LINE__)`, where `vp_gcc_inst_trace` is a VPI instrumentation site insertion macro. There were 11 instrumentation sites, totalling 456 802 calls over a run and yielding a trace file over 23 megabytes long. This is more than a thousandfold increase in instrumentation calls over the `printf()` scenario. We did not run this scenario under QEMU in light of the prohibitive simulation times for the much simpler `printf()` scenario.

Results are detailed in Table V. We only present runtime and simulation time overhead results, since space overhead is negligible in runtime-dominated profiling use cases. As with the `printf()` semi-hosting scenario, large differences exist in results depending on the configuration used. With internal configuration, the instrumentation calls penalized simulation time on the order of 150 μs per call. In contrast, the negative impact on simulation speed of accessing VP state through an external interface is clearly demonstrated by over-

Table V
OVERHEADS PER CALL FOR PROFILING SCENARIO

Instrumentation case	Total simulation time (s)	Runtime overhead (cycles)	Simulation overhead (seconds)
None	0.250	**0**	**0**
Internal VPI	62.59	2.13	136.5μ
Internal stub-call	69.88	9.7	152.4μ
External VPI	3286	4.06	7193μ
External stub-call	3299	9.7	7221μ

head results around 7 ms per call, which is close to 50 times worse than with internal cases. On the opposite end of the performance spectrum, the internal VPI instrumentation case displays significantly lower runtime overhead than the traditional stub-call approach, for a comparable simulation time.

In terms of target runtime overhead, a reduction of 2–5 times over the stub-call case is seen with the VPI cases. If complex behavior had been implemented in the instrumentation functions on the target instead of wrapping a VPI call, overhead would have increased proportionately over the simple stub-call cases shown.

V. CONCLUSION AND FUTURE WORK

Compared to existing semihosting and profiling instrumentation approaches, our contributed framework is shown to have lower runtime and space overhead on the target. In both case study scenarios, our method showed 2–11 times lower runtime interference compared to traditional methods. The lower overall target overhead and the construction of our VPI instrumentation insertion method enable the use of our framework to unify the implementation of previously separate semihosting and tracing/profiling use cases.

Because our method allows for inlining, is fully compatible with all optimization levels and has low target space and time overhead, it may remain in release code. With interception disabled in the VP, instrumented sites do not affect the runtime. This opens the possibility of distributing instrumented binaries which can later be pulled from the field for re-execution with instrumentation enabled under a VP.

In terms of simulation time, our VPI implementation has performances comparable to traditional stub-call semihosting when using the internal configuration.

We have also shown that our framework can be used to extend the instrumentation capabilities of existing VPs without changing their source code. This "add-on instrumentation" capability exploits scripting interfaces currently available in VPs and provides users with the option of reusing our sample implementation in their own environments.

While our results validate our assertions, we must also acknowledge that our prototype implementation suffers from some performance issues which are unrelated to the core VPI concepts presented in this paper.

Firstly, simulation time overhead is dominated by choice of VPI configuration, with the external configuration executing as much as 50 times slower than internal configurations. In the case of our GDB-based external implementation, performances are limited by the communication and context switching overheads between GDB and the VP. These performance issues are due to the architecture of GDB and shared by any tool employing GDB as a generic interface to a virtual platform.

Secondly, since our prototype implementation uses pure Python scripting code, it is at least an order of magnitude slower than what could be achieved using a native C/C++ implementation.

Future work includes implementing our VPI framework on a wider variety of VPs and architectures. Additional case studies and benchmarks could be beneficial in identifying more use cases where our method is an optimization of existing practices, while also serving as validation that inlined instrumentation is robust under more optimizations than those we validated.

ACKNOWLEDGEMENTS

We would like to thank L. Moss, J. Engblom, G. Beltrame and L. Fossati for providing us with valuable insights about code instrumentation on virtual platforms, which helped shape the construction of our framework and its presentation in this paper. We also wish to thank J-P. Oudet and the peer reviewers for helpful comments about the original manuscript.

REFERENCES

[1] Freescale Semiconductors, Inc. (2008, Jun.) Virtutech announces breakthrough hybrid simulation capability allowing mixed levels of model abstraction. Accessed 6/7/2010. [Online]. Available: http://goo.gl/UErXR

[2] Synopsys, inc., *CoWare Platform Architect Product Family: SystemC Debug and Analysis User's Guide*, v2010.1.1 ed., Jun. 2010.

[3] G. Beltrame, L. Fossati, and D. Sciuto, "Resp: A nonintrusive transaction-level reflective mpsoc simulation platform for design space exploration," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 28, no. 12, pp. 1857–1869, Dec. 2009.

[4] L. Moss, M. de Nanclas, L. Filion, S. Fontaine, G. Bois, and M. Aboulhamid, "Seamless hardware/software performance co-monitoring in a codesign simulation environment with rtos support," in *Proc. Design, Automation Test in Europe Conf. Exhibition (DATE)*, 2007, pp. 1–6.

[5] S. Fischmeister and P. Lam, "On time-aware instrumentation of programs," in *Proc. 15th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, Apr. 2009, pp. 305–314.

[6] ARM Ltd, *ARM Compiler toolchain: Developing Software for ARM Processors*, 2010, version 4.1, document number ARM DUI 0471B. [Online]. Available: http://goo.gl/qlKkO

[7] J. Engblom, D. Aarno, and B. Werner, *Full-System Simulation from Embedded to High-Performance Systems*. Springer US, 2010, ch. 3, pp. 25–45. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-6175-4_3

[8] Qemu open-source processor emulator. [Online]. Available: http://www.qemu.org

[9] Imperas Ltd. (2010) Technology ovpsim. Accessed 12/15/2010. [Online]. Available: http://www.ovpworld.org/technology_ovpsim.php

[10] IBM Corporation, *IBM XL C/C++ for Linux, V11.1, Optimization and Programming Guide*, 2010, document number SC23-8608-00. [Online]. Available: http://goo.gl/e1Ri9

[11] J. Corbet. (2007, aug) Kernel markers. Accessed 12/10/2010. [Online]. Available: http://lwn.net/Articles/245671/

[12] B. L. Titzer and J. Palsberg, "Nonintrusive precision instrumentation of microcontroller software," *ACM SIGPLAN Not.*, vol. 40, pp. 59–68, June 2005. [Online]. Available: http://doi.acm.org/10.1145/1070891.1065919

[13] Tensilica, inc., *Xtensa Instruction Set Architecture: Reference Manual*, Santa Clara, CA, Nov. 2006, document number PD-06-0801-00.

[14] H. Shen and F. Petrot, "A flexible hybrid simulation platform targeting multiple configurable processors soc," in *Proc. 15th Asia and South Pacific Design Automation Conf.*, Jan. 2010, pp. 155–160.

[15] N. Anastopoulos, K. Nikas, G. Goumas, and N. Koziris, "Early experiences on accelerating dijkstra's algorithm using transactional memory," in *Proc. IEEE Int. Symp. on Parallel Distributed Processing (IPDPS)*, May 2009, pp. 1–8.

[16] Free Software Foundation. The gnu c compiler. Accessed 11/1/2010. [Online]. Available: http://gcc.gnu.org

[17] S.-S. Lim. (1996) Snu-rt benchmark suite for worst case timing analysis. Original SNU site now down. [Online]. Available: http://www.cprover.org/goto-cc/examples/snu.html

LISTE DE RÉFÉRENCES BIBLIOGRAPHIQUES

[1] RTCA: 1992. *DO-178B : Software considerations in airborne systems and equipment certification*.

[2] AMD and Cyrix: octobre 1995. *The Open Programmable Interrupt Controller (PIC) Register Interface Specification Revision 1.2*.

[3] RTCA: 2001. *DO-248B : Final Annual Report For Clarification Of DO-178B "Software Considerations In Airborne Systems And Equipment Certification"*.

[4] *The KISS-FFT Library Homepage*. En ligne, 2010. `http://kissfft.sourceforge.net/`. Consulté le 23 mai 2011.

[5] *Basic Compression Library*. En ligne, 2011. `http://bcl.comli.eu/`. Consulté le 23 mai 2011.

[6] Adams, Charlotte: *Product Focus : COTS Operating Systems : Boarding the Boeing 787*. En ligne, avril 2005. `http://goo.gl/bbtD9`. Consulté le 23 novembre 2009.

[7] Aeroflex Gaisler AB: *LEON3 RTAX Block Diagram*. En ligne, 2008. `http://www.gaisler.com/doc/leon3ft-rtax.gif`. Consulté le 9 février 2011.

[8] AIA, GAMA et FAA Aircraft Certification Services: *The FAA and Industry Guide to Product Certification*. Federal Aviation Administration, 800 Independence Avenue, SW, Washington, DC 20591, 2e édition, septembre 2004. `http://www.faa.gov/aircraft/air_cert/design_approvals/media/CPI_guide_II.pdf`.

[9] Allerton, D.J.: *High integrity real-time software*. Journal of Aerospace Engineering of the Institution of Mechanical Engineers, 221(G1) :145 – 161, 2007, ISSN 0954-4100. `http://dx.doi.org/10.1243/09544100JAERO60`.

[10] ARINC, Inc: novembre 1997. *ARINC-651 : Design Guidance for Integrated Modular Avionics*. ARINC, Inc, Annapolis, MD. `https://www.arinc.com/cf/store/catalog_detail.cfm?item_id=20`.

[11] ARINC, Inc: juin 2005. *Aircraft Data Network, Part 7, Avionics Full Duplex Switched Ethernet (AFDX) Network*. ARINC, Inc, Annapolis, MD. `https://www.arinc.com/cf/store/catalog_detail.cfm?item_id=574`.

[12] ARINC, Inc: mars 2006. *ARINC653P1 : Avionics Application Software Standard Interface, Part 1- Required Services*. ARINC, Inc, Annapolis, MD. https://www.arinc.com/cf/store/catalog_detail.cfm?item_id=632.

[13] Asanovic, Krste, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel et Katherine Yelick: *A View of the Parallel Computing Landscape*. Communications of the ACM, 52(10) :56–67, octobre 2009.

[14] Barham, Paul, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt et Andrew Warfield: *Xen and the art of virtualization*. Dans *19th ACM symposium on Operating systems principles (SOSP)*, pages 164–177, New York, NY, USA, 2003. ACM, ISBN 1-58113-757-5.

[15] Beltrame, G., L. Fossati et D. Sciuto: *ReSP : A Nonintrusive Transaction-Level Reflective MPSoC Simulation Platform for Design Space Exploration*. IEEE Transactions in Computer-Aided Design of Integrated Circuits and Systems, 28(12) :1857–1869, décembre 2009, ISSN 0278-0070.

[16] Briere, Dominique, Christian Favre et Pascal Traverse: *Electrical Flight Controls, From Airbus A320/330/340 to Future Military Transport Aircraft : A Family of Fault-Tolerant Systems*. Dans Spitzer, Cary R. (rédacteur) : *Digital Avionics Handbook*, tome 1, chapitre 24, pages 24–1 – 24–15. Taylor & Francis Group LLC, 2007.

[17] Budihal, Ramachandra: *Emerging trends in embedded systems and applications*. En ligne, juillet 2010. http://www.eetimes.com/discussion/other/4204667/Emerging-trends-in-embedded-systems-and-applications. Consulté le 29 juillet 2010.

[18] Carmel-Veilleux, Tennessee, Jean François Boland et Guy Bois: *A Novel Low-Overhead Flexible Instrumentation Framework for Virtual Platforms*. Dans *22nd IEEE International Symposium on Rapid System Prototyping (RSP)*, pages 92–98. IEEE, mai 2011.

[19] Crespo, A., I. Ripoll et M. Masmano: *Partitioned Embedded Architecture Based on Hypervisor : The XtratuM Approach*. Dans *European Dependable Computing Conference (EDCC)*, pages 67–72, avril 2010.

[20] Crespo, A., I. Ripoll, M. Masmano, P. Arberet et J.J. Metge: *Xtratum : An open source hypervisor for tsp embedded systems in aerospace*. Dans *Data Systems In Aerospace Conference*, 2009.

[21] Cullmann, Christoph, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet et Reinhard Wilhelm: *Predictability Considerations in the Design of Multi-Core Embedded Systems*. Dans *Embedded Real Time Software and Systems Conference (ERTSS)*, pages 36–42, mai 2010. http://www.erts2010.org/Site/0ANDGY78/Fichier/PAPIERS%20ERTS%202010/ERTS2010_0049_final.pdf.

[22] Delange, Julien: *POK Project Homepage*. En ligne, 2011. http://pok.safety-critical.net. Consulté le 21 mars 2011.

[23] Downey, Allen B.: *The little book of semaphores*. Green Tea Press, 2008. http://greenteapress.com/semaphores/.

[24] Duranton, M.: *The Challenges for High Performance Embedded Systems*. Dans *9th EUROMICRO Conference on Digital System Design : Architectures, Methods and Tools*, pages 3–7, août 2006.

[25] Engblom, Jakob, Daniel Aarno et Bengt Werner: *Full-System Simulation from Embedded to High-Performance Systems*, chapitre 3, pages 25–45. Springer US, 2010, ISBN 978-1-4419-6175-4. http://dx.doi.org/10.1007/978-1-4419-6175-4_3.

[26] Favre, C.: *Fly-by-wire for commercial aircraft : the Airbus experience*. International Journal of Control, 59(1) :139–157, 1994.

[27] Freescale Semiconductors, inc.: *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture*, septembre 2005. Rev 3.

[28] Freescale Semiconductors, inc.: *e600 PowerPC Core Reference Manual*, mars 2006. Rev 0.

[29] Freescale Semiconductors, inc.: *MPC8641D Integrated Host Processor Family Reference Manual*, juillet 2008. http://www.freescale.com/files/32bit/doc/ref_manual/MPC8641DRM.pdf?fpsp=1, Rev 2.

[30] Fuchsen, Rudolph: *How to address certification for multi-core based IMA platforms : Current status and potential solutions*. Dans *IEEE/AIAA 29th Digital Avionics Systems Conference (DASC)*, pages 5.E.3–1–5.E.3–11, octobre 2010.

[31] Goldberg, Robert P.: *Architectural Principles for Virtual Computer Systems*. Thèse de doctorat, Harvard University, Cambridge, MA 02138, février 1973. http://www.dtic.mil/cgi-bin/GetTRDoc?AD=AD772809&Location=U2&doc=GetTRDoc.pdf.

[32] Herlihy, Maurice et Nir Shavit: *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[33] Hilderman, Vance et Tony Baghai: *Avionics Certification : A Complete Guide to DO-178 (Software) and DO-254 (Hardware)*. Avionics Communications Inc., Lessburg, VA, 1re édition, 2007.

[34] International Business Machines Corporation: *The PowerPC Architecture : A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, Inc, 2e édition, 1994.

[35] Itier, Jean Bernard: *A380 Integrated Modular Avionics*. Dans *ARTIST2 meeting on Integrated Modular Avionics*, novembre 2007.

[36] Johnson, K. et R. Saha: *Secure partitioning for multi-core systems*. En ligne, mai 2007.
`http://www.qnx.com/download/download/16559/qnx_secure_partitioning_for_multicore_paper_RIM_MC411.54.pdf`. Consulté le 30 juillet 2010.

[37] Kinnan, L.M.: *Use of multicore processors in avionics systems and its potential impact on implementation and certification*. Dans *IEEE/AIAA 28th Digital Avionics Systems Conference (DASC)*, pages 1.E.4–1–1.E.4–6, octobre 2009.

[38] Klein, Gerwin, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch et Simon Winwood: *seL4 : formal verification of an operating-system kernel*. Communications of the ACM, 53 :107–115, juin 2010, ISSN 0001-0782.
`http://doi.acm.org/10.1145/1743546.1743574`.

[39] L4Ka Team: *L4Ka::Pistachio microkernel*. En ligne, 2011.
`http://www.l4ka.org/65.php`. Consulté le 21 mars 2011.

[40] Liedtke, Jochen: *On micro-kernel construction*. Dans *15th ACM symposium on operating systems principles (SOSP)*, pages 237–250. ACM, 1995, ISBN 0-89791-715-4.
`http://doi.acm.org/10.1145/224056.224075`.

[41] Littlefield-Lawwill, J. et L. Kinnan: *System considerations for robust time and space partitioning in Integrated Modular Avionics*. Dans *IEEE/AIAA 27th Digital Avionics Systems Conference (DASC)*, pages 1.B.1–1–1.B.1–11, 2008.

[42] Mahapatra, Rabi N., Praveen Bhojwani et Jason Lee: *Microprocessor Evaluations for Safety-critical, Real-time Applications : Authority for Expenditure No. 43 Phase 2 Report*. Technical Report DOT/FAA/AR-08/14, Federal Aviation Administration, Washington, DC 20591, juin 2008.

[43] Mahapatra, Rabi N., Praveen Bhojwani, Jason Lee et Yoonjin Kim: *Microprocessor Evaluations for Safety-critical, Real-time Applications : Authority for Expenditure No. 43 Phase 3 Report*. Technical Report DOT/FAA/AR-08/55, Federal Aviation Administration, Washington, DC 20591, février 2009.

[44] Mahapatra, Rabi N., Jason Lee, Nikhil Gupta et Bob Manners: *Microprocessor Evaluations for Safety-critical, Real-time Applications : Authority for Expenditure No. 43 Phase 4 Report*. Technical Report DOT/FAA/AR-10/21, Federal Aviation Administration, Washington, DC 20591, septembre 2010.

[45] Masmano, Miguel, Ismael Ripoll et Alfons Crespo: *An overview of the XtratuM nanokernel*. Dans *1st Intl. Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT)*, 2005, ISBN 121. `http://www.xtratum.org/doc/papers/xtratum_overview_OSPERT2005.pdf`.

[46] Masmano, Miguel, Ismael Ripoll, Alfons Crespo et Vicent Brocal: *XtratuM Hypervisor for LEON2, Volume 2 : User Manual*. Universidad Politecnica de Valencia, 2009.

[47] McIntyre, Geoffrey R.: *The application of system safety engineering and management techniques at the US Federal Aviation Administration (FAA)*. Safety Science, 40(1-4) :325 – 335, 2002, ISSN 0925-7535. `http://www.sciencedirect.com/science/article/B6VF9-445GKC2-K/2/a78441c1cb031f3466453065c2ba67ca`.

[48] Mellor-Crummey, John M. et Michael L. Scott: *Algorithms for scalable synchronization on shared-memory multiprocessors*. ACM Transactions on Computer Systems, 9(1) :21–65, 1991, ISSN 0734-2071.

[49] Moir, Ian et Allan Seabridge: *Military Avionics Systems*. John Wiley & Sons, 2006. `http://www.knovel.com/web/portal/browse/display?_EXT_KNOVEL_DISPLAY_bookid=1428`.

[50] Open Kernel Labs: *OKL4 Microvisor*. En ligne, 2011. `http://www.ok-labs.com/products/okl4-microvisor`. Consulté le 21 mars 2011.

[51] Paolieri, Marco, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat et Mateo Valero: *Hardware support for WCET analysis of hard real-time multicore systems*.

Dans *36th annual international symposium on Computer architecture*, pages 57–68, New York, NY, USA, 2009. ACM, ISBN 978-1-60558-526-0. http://doi.acm.org/10.1145/1555754.1555764.

[52] Patterson, David A. et John L. Hennessy: *Computer Organization and Design : The Hardware/Software Interface*. Morgan Kaufmann, 3e édition, 2005.

[53] Power.org: *Power ISA Version 2.04*. International Business Machines Corporation, 2007.

[54] Power.org: *Power ISA Version 2.06 Revision B*. International Business Machines Corporation, 2010. http://www.power.org/resources/downloads/PowerISA_V2.06B_V2_PUBLIC.pdf.

[55] Ramsey, James W.: *Integrated Modular Avionics : Less is More*. En ligne, février 2007. http://www.aviationtoday.com/av/categories/commercial/8420.html. Consulté le 29 juillet 2010.

[56] Rufino, José et Sérgio Filipe: *AIR - ARINC 653 Interface in RTEMS - Final report*. Technical report, University of Lisbon, décembre 2007. http://air.di.fc.ul.pt/air/downloads/07-35.pdf.

[57] Rushby, John: *Partitioning for Avionics Architectures : Requirements, Mechanisms, and Assurance*. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, juin 1999. http://www.csl.sri.com/users/rushby/papers/faaversion.pdf.

[58] Schoofs, Tobias: *AIR - ARINC 653 Interface in RTEMS - WP4 : Final report*. Technical report, Skysoft Portugal, novembre 2009.

[59] Solihin, Yan: *Fundamentals of Parallel Computer Architecture : Multichip and Multicore Systems*. Solihin Publishing & Consulting LLC, 2009.

[60] SPARC International, Inc.: 1992. *The SPARC Architecture Manual Version 8*. SPARC International, Inc., 535 Middlefield Rd, Suite 210, Menlo Park, CA.

[61] Spradlin, Richard E.: *Modern air transport flight deck design*. Displays, 8(4) :171–182, 1987, ISSN 0141-9382. http://www.sciencedirect.com/science/article/B6V01-47X767R-15/2/75371db4f1ea41c4d0c48774dd30e82c.

[62] SYSGO AG: *Press release : SYSGO integrates flexible multi-core support in new DO-178B certified PikeOS 3.1 release*. En ligne, mars 2010. http://goo.gl/SHnhU. Consulté le 30 juillet 2010.

[63] Titzer, Ben L. et Jens Palsberg: *Nonintrusive precision instrumentation of microcontroller software*. ACM SIGPLAN Notices, 40 :59–68, juin 2005, ISSN 0362-1340. http://doi.acm.org/10.1145/1070891.1065919.

[64] Ungerer, Theo, Francisco Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quinones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Casse, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Floria Kluge, Stefan Metzlaff et Jorg Mische: *Merasa : Multicore Execution of Hard Real-Time Applications Supporting Analyzability*. IEEE Micro, 30 :66–75, 2010, ISSN 0272-1732.

[65] Vestal, Stephen C., Pamela Binns, Aaron Larson, Murali Rangarajan et Ryan Roffelsen: *Safe partition scheduling on multi-core processors*. Brevet en instance, août 2010. http://www.freepatentsonline.com/y2010/0199280.html. Consulté le 21 mars 2011.

[66] Wahbe, Robert, Steven Lucco, Thomas E. Anderson et Susan L. Graham: *Efficient software-based fault isolation*. Dans *14th ACM Symposium on operating systems principles (SOSP)*, pages 203–216. ACM, 1993, ISBN 0-89791-632-8.

[67] Walter, Randy: *Flight Management Systems*. Dans Spitzer, Cary R. (rédacteur) : *Digital Avionics Handbook*, tome 1, chapitre 20, pages 20–1 – 20–26. CRC Press, 2007. http://dx.doi.org/10.1201/9780849384394.ch20.

[68] Walter, Randy et Chris Watkins: *Genesis Platform*. Dans Spitzer, Cary R. (rédacteur) : *Digital Avionics Handbook*, tome 2, chapitre 12, pages 12–1–12–28. Taylor & Francis Group LLC, 2007.

[69] Wind River, Inc: *ARINC 653 An Avionics Standard for Safe, Partitioned Systems*. En ligne, juin 2008. http://www.computersociety.it/wp-content/uploads/2008/08/ieee-cc-arinc653_final.pdf. Consulté le 22 avril 2009.

[70] World Wide Web Consortium: *XML Schema Part 0 : Primer*. Rapport technique 2nd Ed, octobre 2004. http://www.w3.org/TR/xmlschema-0/.