Université de Montréal

Intégration d'un système d'exploitation dans le flot de développement logiciel/matériel

par
Marc Julien

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures et postdoctorales
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

décembre, 2008

Université de Montréal
Faculté des études supérieures et postdoctorales

Ce mémoire intitulé:

Intégration d'un système d'exploitation dans le flot de développement logiciel/matériel

présenté par:

Marc Julien

a été évalué par un jury composé des personnes suivantes:

Fabian Bastin,	président-rapporteur
El Mostapha Aboulhamid,	directeur de recherche
Stefan Monnier,	membre du jury

Mémoire accepté le: ....20/04/2009....

RÉSUMÉ

Les systèmes informatiques devenant de plus en plus complexes, il faut sans cesse chercher des façons d'accélérer le cycle de développement. Comment passer d'un niveau d'abstraction élevé à un autre plus bas, que nous savons déjà traiter, en minimisant les interventions humaines ?

Ma recherche tente de trouver un flot de développement qui permettrait d'abstraire le système d'exploitation afin de repousser le moment de la sélection finale du système d'exploitation. Le flot proposé tente aussi d'automatiser autant que possible les différentes tâches de synthèse du modèle de haut niveau afin de faciliter l'exploration des possibilités de partitionnement et de choix de systèmes d'exploitation.

Pour y arriver, je propose d'expérimenter en utilisant un système d'exploitation léger et simple, Mutek, et un autre plus lourd et complexe, GNU/Linux. Le tout s'exécute sur une plateforme AMIRIX AP1100 possédant un FPGA Virtex 2 Pro muni de deux processeurs PowerPC. Nous explorerons aussi d'autres choix de systèmes d'exploitation et de langage source pour le modèle de haut niveau.

Mots clés: co-design,OS,Virtex2P,compilation,FPGA.

ABSTRACT

Embedded system becoming increasingly complex, we should try to develop new ways to increase the efficiency of the design process. How can we translate our system from a high level of abstraction to another one that we can synthesize while reducing the needs for human intervention?

My master's thesis attempt to find a workflow to abstract the operating system so that the time of final selection of the operating system can be delayed. The suggested flow also tries to maximise the synthesis automation of the high level model. This will help the exploration of both the design space and the operating system.

To achieve this, I used two different operating systems: Mutek, a small and simple one, and GNU/Linux, which is much larger and much more complex. I experimented this on an Amirix AP1100 board which contain a Virtex 2 pro FPGA with two PowerPC processors. I will also explore other possibility of operating system and some possibility of programming language for the high level model.

Keywords: codesign,OS,Virtex2P,compilation,FPGA

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

LISTE DES ABRÉVIATIONS

API	Application Programming Interface (Interface de programmation d'application)
BRAM	Block Random Access Memory (Blocs de mémoire de Xilinx)
DDR	Double Data Rate (Débit de données double)
DPN	Dysident Process Network (Réseau de processus Dysident)
DSP	Digital Signal Processing (Circuit optimisé pour le traitement de signal)
EDA	Electronic Design Automation (Automatisation de la conception électronique)
ESL	Electronic System Level (Niveau système électronique)
FIFO	First In, First Out (Premier entré, premier sorti)
FPGA	Field Programmable Gate Array (Puce contenant de la logique reprogrammable)
IC	Integrated Circuit (Circuit Intégré)
ISS	Instruction Set Simulator (Simulateur de jeux d'instructions)
ITRON	Industrial « The Real-time Operating system Nucleus » (API standardisé)
OPB	On-chip Peripheral Bus (Bus périphérique du CoreConnect (c) d'IBM)
PLB	Processor Local Bus (Bus processeur du CoreConnect (c) d'IBM)
POSIX	Portable Operating System Interface (API standardisé)
RTL	Register Transfer Level (Niveau transfert de registre)
RTOS	Real-Time Operating System (Système d'exploitation temps réel)
SDRAM	Synchronous Dynamic Random Access Memory (Type de mémoire vive)
UART	Universal Asynchronous Receiver Transmitter (Port série)
UDI	Uniform Driver Interface (Interface uniforme pour pilotes de périphériques)
VHDL	Very high speed integretad circuits Hardware Description Language (Langage de description matériel)

REMERCIEMENTS

Je voudrais remercier M. El Mostapha Aboulhamid, Ph.D. pour m'avoir permis de réaliser ma recherche au sein de son laboratoire.

Je voudrais également remercier mes parents, mes amis et ma copine Magali de leur soutien et de leurs encouragements.

Des remerciements spéciaux à Mathieu Desnoyers et Paul Khuong pour m'avoir relu et corrigé.

CHAPITRE 1

INTRODUCTION

Le développement des systèmes informatiques a beaucoup évolué depuis ses débuts, passant de pièces électroniques discrètes à des circuits imprimés, puis aux circuits intégrés. La complexité de ces circuits s'est, elle aussi, grandement accentuée. Pour obtenir les mêmes capacités qu'un ordinateur immense lors des premières années de l'informatique, il faut maintenant compter quelques fractions de millimètres. De plus, un simple téléphone contient maintenant plusieurs fois la capacité de calcul de ces mastodontes. Nous ne pouvons donc absolument plus utiliser les mêmes moyens de conception qu'à l'époque. Des méthodes plus efficaces ont donc vu le jour. De nos jours, nous concevons le matériel à peu près de la même façon que nous concevions le logiciel à l'époque, soit avec des langages de bas niveau qui donnent beaucoup de contrôle, mais qui , comparés à des langages de plus haut niveau, ralentissent le développement de nouvelles applications et sont plus sujets à erreurs. De plus, les architectures reconfigurables rendent de plus en plus accessible la création matérielle et en accélèrent encore davantage le développement.

Il devient donc de plus en plus important d'avoir un flot de conception de haut niveau qui nous permet d'accélérer le développement d'un système matériel. Une idée souvent étudiée est d'utiliser un langage de haut niveau qui se rapproche de ceux utilisés de nos jours pour l'écriture de logiciel. Plusieurs problèmes se posent cependant avec cette technique. Premièrement, comment peut-on transformer un code de haut niveau avec des appels de fonctions et peut-être même de la récursion en un ensemble de fils et de transistors ? Comment est-il possible de s'assurer que le système produit est bien conforme à celui décrit dans notre langage de haut niveau ?

Cette approche est souvent utilisée en conjonction avec une technique nommée le codesign ou conception mixte. Cette technique consiste à tenter de produire la couche matérielle en même temps que celle logicielle. Cette approche, en plus d'accélérer le développement d'architectures matérielles, offre l'avantage de pouvoir tester le logiciel

avant la mise en place de l'architecture matérielle finale. De plus, cette façon de faire permet d'effectuer diverses mesures afin de trouver le compromis matériel/logiciel correspondant à nos besoins, tant en terme de coût que de performance d'exécution. Cette phase est nommée exploration de l'espace de conception. Ce mode de conception est particulièrement bien adapté à la réalité des systèmes embarqués, puisque ces circuits doivent réaliser des tâches précises avec un environnement spécifique. De nos jours, ces systèmes embarqués se retrouvent partout : dans les automobiles, les avions, les centrales nucléaires et même les téléphones. Ils sont aussi utilisés à l'intérieur de systèmes plus complexes. Ainsi, la plupart des cartes d'expansion pour ordinateurs peuvent être considérées comme des systèmes embarqués. Les systèmes embarqués ont, dans la grande majorité des cas, de fortes restrictions en terme de puissance de calcul et de mémoire disponible. Il faut donc, pour satisfaire ces limitations, porter une attention toute particulière aux logiciels utilisés.

Le système d'exploitation étant le maître qui contrôle tous les autres logiciels, son choix s'avère souvent crucial dans le développement, le déploiement et la maintenance d'un système. Chaque système d'exploitation ayant ses capacités, ses points forts et ses points faibles, il serait intéressant de pouvoir facilement passer d'un système d'exploitation à un autre afin de tester lequel nous convient le mieux. Il pourrait aussi être intéressant de pouvoir utiliser les outils disponibles sous un système d'exploitation lors du développement pour ensuite utiliser un système d'exploitation plus léger ou sans redevances pour le déploiement.

1.1 Objectifs

Mon objectif est de développer un flot de développement mixte logiciel/matériel qui repousserait le plus possible le choix final du système d'exploitation sans pour autant retarder le développement du matériel, ni celui du logiciel. Le flot devrait aussi permettre d'effectuer des tests avec plusieurs systèmes d'exploitation différents à chaque étape de la création d'un nouveau système. Puisque le développement mixte se prête particulièrement bien aux systèmes embarqués, mes travaux vont se limiter à ceux-ci. Cela

offre certaines simplifications, mais force aussi à travailler dans des environnements restreints, notamment au niveau de la mémoire vive disponible et de la capacité de calcul. Bien qu'une bonne partie des systèmes embarqués nécessitent des primitives temps réel, cela entraine plusieurs questionnements supplémentaires qui ne seront pas posés dans ce mémoire.

1.2 Approche suggérée

Selon le EETimes de janvier 2007[14], les principales compagnies de conception électronique assistée par ordinateur (*electronic design automation* ou EDA) essaient de développer autant que possible les techniques de développement au niveau système électronique (*electronic system level* ou ESL). Cependant, toujours selon le même article, il existe encore certains trous dans la méthodologie. L'approche que je suggère consiste à prendre un programme dans un langage fixé, idéalement de haut niveau, et de le traduire en un système logiciel/matériel sur une plateforme donnée. Cette approche est assez répandue dans le domaine de la conception mixte logiciel/matériel. Cependant, lors de la construction de ce flot, je tâcherai de porter une attention toute particulière à la place du système d'exploitation afin de repousser le moment de la sélection finale de celui-ci. Plusieurs problèmes se posent pour la réalisation de cette tâche et il faudra donc, vu la complexité, restreindre la visée.

Un flot de développement d'un système combinant logiciel et matériel serait donc de développer une application réalisant les fonctions désirées dans un langage de haut niveau. Nous ne pourrons probablement pas supporter le langage source au complet, il faudra alors que le programme respecte certaines règles, notamment sur les fonctions standard supportées ou sur la structure des programmes. Ensuite, il faudrait spécifier, avec un fichier de configuration, quels modules ou fonctions on désire avoir en matériel et lesquelles doivent rester exécutées en logiciel. Ce fichier contiendrait aussi quelques informations concernant le ou les systèmes d'exploitation que l'on désire utiliser ainsi que de l'information sur le matériel externe et les connexions d'entrée/sortie. Un ensemble de programmes prendrait alors le code de notre application et notre fichier de

configuration et générerait, de la façon la plus automatisée possible, les fichiers de description matérielle et un paquetage logiciel exécutable sur la plateforme spécifiée pour chacun des systèmes d'exploitation désirés.

Dans le présent mémoire, je m'attarderai principalement aux aspects logiciels d'une telle approche et laisserai la synthèse pour des travaux futurs. Je vais donc me concentrer un peu plus spécifiquement sur l'aspect du système d'exploitation dans un tel système de codesign et sur les façons de retarder la sélection du système d'exploitation dans le flot. Je m'attarderai aussi sur les façons d'effectuer la synthèse des communications.

Le flot que je propose permet, si l'on respecte les restrictions imposées, de changer de système d'exploitation pratiquement à n'importe quelle phase de développement et avec très peu d'efforts. Cela est rendu possible surtout grâce à l'utilisation d'interfaces standardisées. De nombreuses améliorations peuvent cependant être ajoutées, notamment au niveau de l'automatisation du processus de création du matériel.

Pour débuter, je vais commencer par faire une revue des développements dans ce secteur. Ensuite, je présenterai la plateforme que nous allons utiliser dans notre investigation. Je ferai, par la suite, une revue des systèmes d'exploitation embarqués offerts sur le marché, en m'attardant un peu sur les deux que j'ai choisi d'utiliser dans ma recherche : GNU/Linux et Mutek. Après, j'expliquerai mon choix de langage source pour le codesign pour me diriger vers la problématique du mode de communication entre le matériel et le logiciel. Viendra ensuite un exemple d'application que j'ai étudié pour approfondir les problèmes qui pourraient survenir. Je ferai alors un résumé du flot de développement que je propose ainsi que diverses idées pour le compléter et accroître ses capacités.

CHAPITRE 2

REVUE DE LITTÉRATURE

La loi de Moore suggère que le nombre de transistors sur une puce double approximativement tous les deux ans[19]. Certains on prédit certaines limites à cette hausse exponentielle, mais pour l'instant la loi continue de s'imposer[12]. Le nombre de fonctionnalités et l'espace disponible pour du matériel spécialisé s'accroissent donc eux aussi de façon exponentielle. La compétition étant féroce dans le domaine des nouvelles technologies, le temps de l'apparition d'un projet jusqu'à sa mise en marché a, quant à lui, grandement diminué. Il a donc fallu développer des techniques pour accélérer le développement.

2.1 Design traditionnel

2.1.1 Logiciels utilisateurs

Les logiciels utilisateurs sont ceux qui ne communiquent avec le système que par l'entremise du système d'exploitation et n'essaient pas d'accéder directement au matériel. De nombreuses avancées ont eu lieu dans le domaine pour réagir à la demande d'accélération du développement du côté des logiciels utilisateurs. Une approche souvent utilisée est d'augmenter la réutilisation du logiciel, grâce, entre autres, à des interfaces standardisées qui permettent de facilement porter une application vers une nouvelle plateforme. POSIX et μITRON sont des exemples de ces interfaces standardisés utilisés dans le domaine des systèmes embarqués. Une autre approche est d'utiliser un langage de plus haut niveau, ce qui accélère grandement le développement.

Bien que des développements continuent d'avoir lieu pour ces logiciels, les niveaux actuels de réutilisation et de facilité de développement sont grandement supérieurs à ceux observés pour les logiciels systèmes et le matériel.

2.1.2 Logiciels systèmes

Je nomme logiciels systèmes les logiciels qui nécessitent un accès direct au matériel. Ces logiciels fonctionnent habituellement dans un mode particulier du processeur, ce qui leur donne beaucoup plus de pouvoir, mais ces nouvelles possibilités sont rarement supportées par les compilateurs. Dans plusieurs cas, ces logiciels fonctionnent aussi dans un mode d'adressage mémoire particulier qui offre moins de protection contre les erreurs et moins de facilité de débogage que les logiciels utilisateurs. Les systèmes d'exploitation et les pilotes de périphériques sont les principaux logiciels système.

Les améliorations apportées aux logiciels utilisateurs tardent cependant à percer auprès des logiciels systèmes. Ces logiciels, ayant des besoins très particuliers, doivent normalement avoir des interfaces d'assez bas niveau pour pouvoir dialoguer directement avec le matériel. Les performances de ces bouts de code sont aussi souvent très critiques et les conséquences des erreurs de programmation parfois très graves. C'est pourquoi leurs concepteurs sont souvent très réticents à changer leurs habitudes. Cependant, on ne peut nier les avantages qu'apporterait une accélération du développement et une augmentation de leur portabilité entre différents systèmes d'exploitation.

Pour atteindre notre but, nous aurons besoin d'une stratégie pour permettre aux logiciels systèmes, dans notre cas les pilotes de périphériques qui parlent à nos composantes matérielles, de supporter plusieurs systèmes d'exploitation. Certaines recherches ont déjà été effectuées pour arriver à ce but.

Certains essaient de diviser l'écriture des logiciels systèmes en séparant l'interface du système d'exploitation du fonctionnement du logiciel. Cette approche facilite la portabilité, mais la partie propre au système d'exploitation peut nécessiter beaucoup de temps à concevoir. Des tentatives d'uniformiser l'interface avec le système d'exploitation ont eu lieu, par exemple le *Uniform Driver Interface*[2]. Cette approche remonte à quelques années et nécessite l'ajout d'une couche de compatibilité aux systèmes d'exploitation désirant supporter l'interface. Cependant, une fois cette couche établie, tous les pilotes de périphériques développés en utilisant l'interface peuvent fonctionner avec le système d'exploitation. Certains défenseurs des logiciels libres se sont prononcés contre cette

interface puisqu'elle, selon eux, inciterait les fabricants de matériel à développer leurs pilotes en licence propriétaire [25]. L'approche pourrait être intéressante, mais peu de systèmes la supportent. Elle nous offre donc peu d'aide.

D'autres ont essayé d'augmenter le niveau auquel sont normalement développés les pilotes de périphérique et de les développer au niveau objet[17]. L'approche est intéressante, mais nécessite le support d'un langage orienté objet, ce qui est loin d'être acquis pour la plupart des systèmes d'exploitation embarqués actuels. De plus, l'approche orientée objet nécessite habituellement plus d'appels de fonction pour réaliser les tâches, ce qui est problématique pour les systèmes embarqués, qui ont besoin de performances, mais qui sont limités en capacité de calcul.

Certaines recherches explorent aussi la possibilité d'avoir une spécification du logiciel système écrite dans un langage spécialisé et permettant de générer une implémentation qui correspond au système d'exploitation utilisé[15][20]. Cette approche nécessite aussi beaucoup de travail pour supporter un nouveau système d'exploitation et l'apprentissage d'un nouveau langage de la part du concepteur.

2.1.3 Matériel

Les premiers ordinateurs ont été conçus à l'aide de composantes discrètes, c'est-à-dire que chaque puce ne contenait que quelques transistor (Small Scale Integration ou SSI) ou centaines (Medium Scale Integration ou MSI) qu'il fallait interconnecter à la main. De nos jours, une puce peut contenir des milliards de ces transistors (Very Large Scale Integration ou VLSI). La conception du matériel a donc aussi fait du chemin, mais est encore loin d'être aussi accessible que le développement de logiciels utilisateurs. Bien sûr quelques systèmes offrent une interface graphique pour brancher des composantes ensemble, mais cela revient presque à utiliser des composantes discrètes. Les logiciels permettent donc plusieurs niveaux d'abstraction, allant même jusqu'à proposer une interface à laquelle nous pouvons ajouter des bus, puis des processeurs et quelques autres composantes d'entrée/sortie et voilà un nouveau système de conçu. Cependant, il reste difficile de concevoir de nouvelles composantes. Les concepteurs essaient donc d'augmenter le niveau d'abstraction. Les méthodes de conception actuelles permettent

de développer du matériel dans des langages basés sur C[22] et C++[21].

2.2 Conception mixte

Comme nous l'avons vu précédemment, la conception mixte consiste à essayer de développer la couche matérielle en même temps que la couche logicielle.

L'avantage de ces méthodologies est de permettre de développer la partie logicielle avant que la partie matérielle ne soit finalisée. Elle permet aussi d'effectuer des tests plus complets sur la logique de la partie matérielle. Certaines approches, comme [3], utilisent des langages différents pour le matériel (VHDL) et pour le logiciel (C). Cette approche fixe malheureusement très tôt le partitionnement entre le logiciel et le matériel, ce qui est indésirable si nous voulons pouvoir tester plusieurs combinaisons. D'autres, comme [8], utilisent un langage de description matériel (Verilog) pour développer le logiciel. Cela facilite le partitionnement, mais nous oblige à abaisser le niveau du langage utilisé pour le logiciel, ce que nous voulons éviter. [7] utilise aussi un langage de descriptions matériel, mais celui-ci est plus haut niveau (basé sur C++) et semble déjà un peu plus approprié.

Étant donné que le matériel sous-jacent est sujet à changement, résoudre les problèmes de portabilité est essentiel pour la conception mixte. Cela est en partie dû au fait qu'idéalement, les parties du projet qui finiront en matériel et celles qui finiront en logiciel ne sont pas fixées au début du projet. Pour résoudre ce problème, les environnements de développement de conception mixte n'utilisent souvent qu'un système d'exploitation fixé[7] et parfois même aucun système d'exploitation[8] conjugué avec un partitionnement figé lors de la synthèse. La façon d'adapter le système à un autre système d'exploitation est cependant rarement spécifiée. Nous désirons avoir un système de conception mixte qui permette l'utilisation de la plus grande diversité de systèmes d'exploitation possible, il faudra donc trouver un moyen d'adapter notre méthodologie afin qu'elle puisse supporter plusieurs systèmes d'exploitation.

2.3 Partitionnement à la volée

La venue des systèmes reconfigurables et la baisse de leurs prix portent les domaines de la synthèse et de la conception mixte vers de nouveaux horizons. Certains travaux sont en cours pour tenter de produire des systèmes dont le partitionnement se fait à la volée[4]. Le partitionnement à la volée consiste à redéfinir quelles parties s'exécutent en matériels pendant l'exécution du programme. Cela ouvre la porte à la définition d'instructions spécialisées pour chaque programme. Un programme effectue de nombreuses multiplications simultanées ? Eh bien consacrons une partie du matériel reconfigurable à cette tâche. Le suivant nécessite des transformées de Fourrier ? Alors, enlevons le matériel de multiplication de matrice pour le remplacer par celui d'une transformée de Fourrier. Les tâches nécessitant une grande capacité de calcul peuvent donc bénéficier de l'accélération procurée par le matériel, sans pour autant avoir les coûts de conception associés : le même matériel peut être réutilisé à plusieurs reprises. Les systèmes reconfigurables actuels permettent même de sauver l'état du matériel dans une mémoire externe, ce qui permet une gestion des tâches concurentes comme on en voit dans le monde du logiciel. Les systèmes de conception mixte devront donc se faire de plus en plus performants et flexibles. Pour pouvoir bénéficier de ces avancées, il faudra donc que toutes les tâches nécessaires à la conception mixte soient automatisées. Pour y arriver, je suggère une approche qui porte une attention particulière au système d'exploitation et qui se veut portable et flexible.

CHAPITRE 3

PLATE-FORME D'ESSAI

3.1 Motivation

Bien que j'aurais aimé que les résultats de ma recherche soient le plus portables que possible, une certaine dose de réalisme s'impose. Le temps étant malheureusement une ressource finie, il convient de restreindre un peu le travail. J'ai donc choisi de restreindre la portabilité de ma recherche en adoptant une plateforme assez précise.

Étant sur un FPGA, la plateforme reste quand même adaptable pour satisfaire nos besoins.

Je vais donc, dans ce chapitre, présenter la plateforme que j'ai choisi d'adopter, en m'attardant un peu plus sur les aspects intéressants. Je vais débuter en présentant brièvement le choix de la carte Amirix AP-1000. Ensuite, je décrirai les ressources physiques et logiques de la plateforme de base proposée. Je présenterai par la suite la répartition de la mémoire pour enfin me diriger vers les différentes configurations possibles qui pourraient nous être utiles.

3.2 Synthèse et FPGA

La plateforme doit nécessairement permettre d'insérer un ou plusieurs composants synthétisés. On appelle synthèse matérielle ou simplement synthèse le fait de partir d'une description dans un langage spécialisé pour arriver à une puce électronique. Traditionnellement, le processus de synthèse nécessite l'impression sur des gaufres de silicium, ce qui est très cher, surtout pour la production d'une petit nombre d'exemplaires.

Heureusement, une technologie relativement nouvelle, les FPGA ou *field programmable gate array*, nous permet d'avoir beaucoup de flexibilité dans la création de matériel et d'abaisser les coûts. Ces circuits intégrés contiennent de la logique reprogrammable qui peut être configurée pour réaliser les circuits combinatoires et séquentiels que l'on veut. Les bornes d'entrée-sortie et les horloges peuvent aussi être routées comme on

le souhaite. La réalisation des circuits logique est faite à l'aide de certaines primitives, notamment des tables de correspondances, des registres et des banques de mémoire. Les tables de correspondances servent à implémenter les circuits combinatoires. Nous pouvons par exemple spécifier que nous voulons que notre table de correspondance retourne un 1 seulement si ses 2 entrées sont à 1, ce qui nous donne un « ET » logique. Les registres servent à retenir une valeur pour plusieurs cycles de l'horloge. Les banques de mémoire servent à retenir un plus grand nombre de données à la manière d'une mémoire RAM. Ces banques de mémoires, appelées *bloc random access memory* ou BRAM , sont organisées en blocs répartis un peu partout sur le FPGA. Plusieurs FPGA contiennent amplement de ces primitives pour permettre de synthétiser des processeurs. Les fabricants de FPGA fournissent donc habituellement un ou plusieurs choix de processeurs maison particulièrement bien adaptés pour leurs FPGA. Xilinx offre les processeurs microblaze et picoblaze, tandis que son compétiteur Altera offre les NIOS et NIOS-II. Il existe même des processeurs à code source ouvert, comme le LEON et le OpenRISC. Lorsque les processeurs sont synthétisés sur un FPGA, on parle de processeurs softcore.

3.3 Choix

Puisqu'il fallait fréquemment changer de configuration, autant pour changer le type et le nombre de processeurs que pour rajouter des blocs matériels avec lesquels communiquer, un FPGA semblait un choix tout désigné. Le FPGA Virtex II-Pro, de la compagnie Xilinx contient deux processeurs de type PowerPC et propose assez de portes logiques pour mettre plusieurs processeurs Microblaze. Puisque nous avions déjà quelques cartes Amirix AP1000 qui incluaient un de ces FPGA (modèle XC2VP100), j'ai décidé de continuer avec cette carte.

La plateforme de base fournie par Amirix contient un processeur PowerPC, un bus PLB, un bus OPB et plusieurs autres blocs de gestion de la mémoire, d'interruptions et d'entrées/sorties (voir section 3.4). Le PLB, pour *Processor Local Bus*, est le type de bus utilisé par les processeurs de type PowerPC pour communiquer avec les composantes rapides du processeur comme la mémoire. Parallèlement, l'OPB, pour *On-Chip Peripheral*

Bus, est utilisé pour communiquer avec les périphériques basse vitesse, comme le port sériel. Donc, la plateforme contient beaucoup de composantes dont nous pourrions avoir besoin et nous pouvons en rajouter au besoin. Nous pourrions, par exemple, y brancher le deuxième PowerPC ou des processeurs Microblaze.

La plateforme convient donc à nos besoins et peut être étendue pour répondre à des besoins ultérieurs.

3.4 Composantes

La carte Amirix contient plusieurs composants et la plateforme de base en définit plusieurs autres synthétisés. Certains sont intéressants pour notre exercice, mais nous en laisserons plusieurs autres de côté. Je vais parler ici seulement de ceux que nous utiliserons dans le présent projet. Le nombre de processeurs PowerPC est limité puisque ceux-ci sont en permanence à l'intérieur du FPGA. Je les inclus dans la section logique, parce qu'ils peuvent être complètement désactivés et nous pouvons effectuer plusieurs configurations lors de la synthèse. Pour de plus amples renseignements sur les composantes disponibles, consultez [1].

Physiques

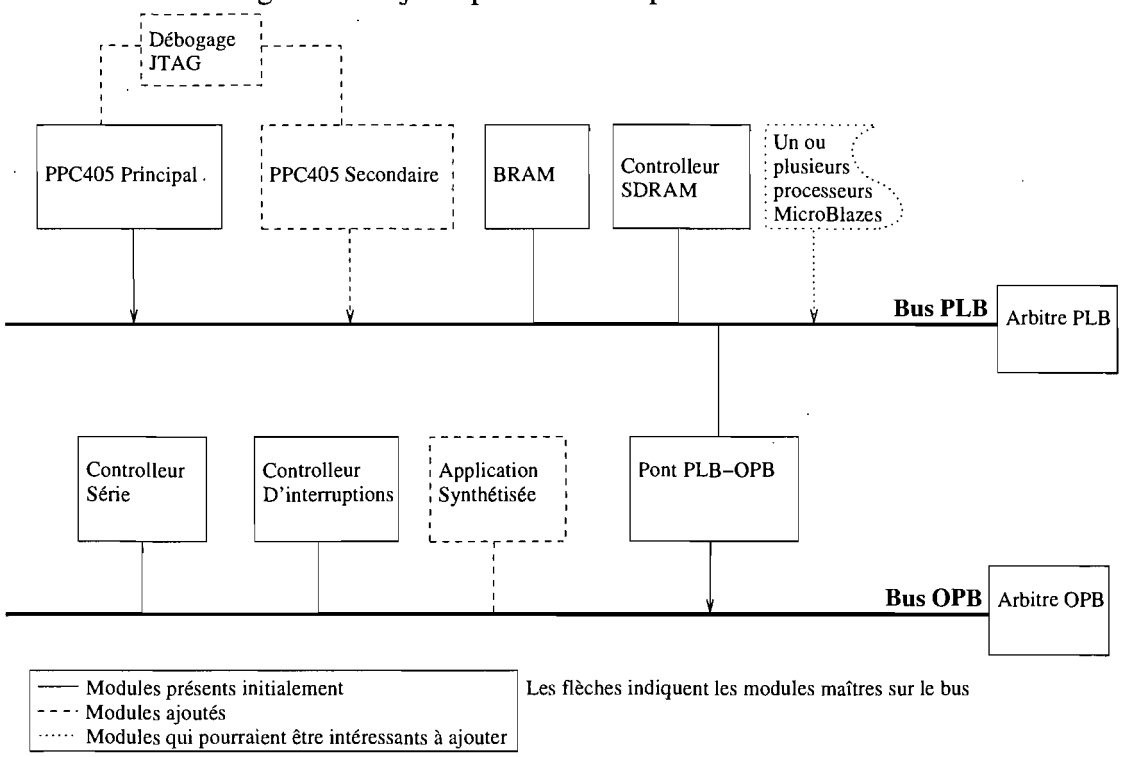
- SDRAM DDR (64 Mo, mémoire principale)
- RS-232 Transceiver (Contrôleur série pour communication avec le monde extérieur)
- E/S d'expansion (Ajout d'extension, par exemple des ports Multi-Gigabit Transceiver ou Ethernet)
- Port JTAG (Programmation du FPGA et débogage)
- SystemACE avec carte CompactFlash (Programmation du FPGA et mémoire secondaire non volatile)
- Mémoire Flash de programme (16 Mo, Mémoire secondaire non volatile)
- Mémoire Flash de configuration (16 Mo, Programmation du FPGA)
- FPGA Xilinx XC2VP100 (Logique reprogrammable)

Reconfigurable (FPGA)

- PowerPC(1 ou 2) (Processeur principal)
- BRAM (Bloc RAM, mémoire volatile initialisée)
- Bus PLB et OPB (Arbitre, pont et autres, communication entre les différents composants)
- Contrôleur pour les périphériques physiques (DDR SDRAM et RS-232)
- Contrôleur d'interruptions

Le schéma 3.1 illustre les composantes logiques de base et quelques extensions possibles.

Figure 3.1: Ajouts possibles à la plateforme de base



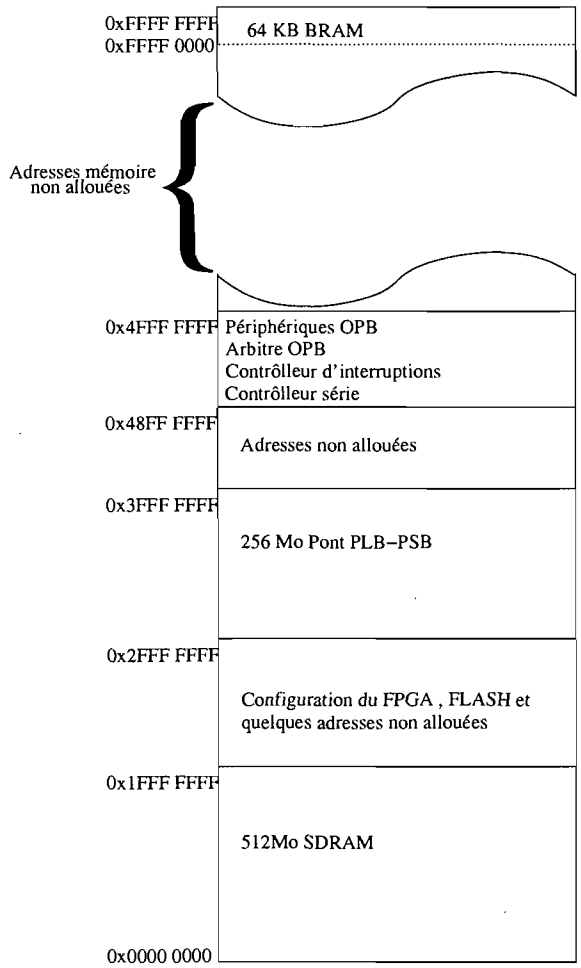
3.5 Mémoire

Comme le démontre la figure 3.2, l'espace d'adressage laisse près de 3 Go inutilisés, ce qui nous permet d'étendre la mémoire physique au besoin ou d'ajouter des périphériques et coprocesseurs directement dans l'espace d'adressage. Nous pouvons aussi constater que l'UART(RS-232) est accessible à l'adresse 0x4C00 0000. Ce composant sera une des principales sources de communication avec le système s'exécutant sur le FPGA. Les autres sources de communication que nous pouvons facilement utiliser sont les lumières DEL et commutateurs, qui ne peuvent pas transmettre une grande quantité d'information, et le port JTAG, qui va nous permettre de contrôler les processeurs afin de pouvoir faire du débogage. D'autres moyens de communication existent, par exemple le port Ethernet, mais sont trop complexes pour être utilisé à moins que le système d'exploitation utilisé offre déjà le support. Le schéma nous présente 512 Mo de SDRAM disponible, mais seulement 64 Mo sont disponibles physiquement. Ces 64 Mo sont donc représentés à plusieurs reprises pour former le 512 Mo.

La plateforme dispose également d'une mémoire BRAM de 64 Ko contenant le code de démarrage du système d'exploitation. La BRAM est initialisée en même temps que le reste du FPGA. On peut donc y insérer des instructions et des données qui seront accessibles dès le démarrage du processeur, mais qui seront remises à leurs valeurs initiales lors du redémarrage de la plateforme. La BRAM contient la première instruction à être exécutée (celle à l'adresse 0xFFFF FFFC dans le cas du PowerPC) et normalement elle contient aussi le code du chargeur qui s'occupe de copier le programme de la mémoire Flash de programme à la mémoire principale et d'y brancher. La taille de la BRAM étant limitée, on ne peut y faire tenir de gros programme ou de système d'exploitation lourd. Cependant, Mutek, un des systèmes d'exploitation que nous avons choisis (voir chapitre 4) étant très petit, on peut le faire tenir dans cet espace si notre application est elle aussi de petite taille.

La mémoire SDRAM est une mémoire volatile et est habituellement utilisée pour contenir le programme une fois qu'il a été chargé depuis la mémoire secondaire. Elle contient habituellement la pile, le tas ainsi que toutes les données de travail des pro-

Figure 3.2: Configuration de la mémoire de la plateforme de base



grammes exécutés, sauf celles qui doivent être persistantes.

La mémoire Flash de programme contient les données qui doivent être sauvegardées d'un démarrage à un autre. La mémoire CompactFlash peut être utilisée au même titre que la mémoire Flash et offre l'avantage de pouvoir être facilement retiré de la carte afin de la modifier à partir d'un autre ordinateur.

Lors d'un démarrage, le processeur branche à l'adresse 0xFFFF FFFF, qui contient habituellement un saut dans la BRAM. Si l'utilisateur n'interrompt pas le processus, un chargeur de démarrage miniature localisé dans la BRAM s'occupe de copier U-Boot de la mémoire Flash de programme à la mémoire SDRAM. U-Boot prend alors la relève. U-Boot est un autre chargeur de démarrage qui est beaucoup plus complet que celui chargé initialement. L'utilisateur peut alors, à l'aide de U-Boot, charger manuellement son programme et brancher sur celui-ci. Le chargement peut se faire de différentes façons, soit par le port sériel, par la carte réseau (protocole tftp) et même par le bus PCI de la machine hôte. Si U-Boot n'est pas interrompu, il exécute une commande contenue dans les variables non volatiles situées dans la mémoire Flash de programme. Par défaut, cette commande charge un noyau Linux et une image de démarrage. Le restant de la distribution GNU/Linux est situé dans la mémoire Flash de programme.

3.6 Configuration

Puisque la base de la plateforme est un FPGA, nous avons beaucoup de choix de configurations possibles. Le nombre de ces configurations étant très grand, nous n'allons nous attarder qu'à une partie de celles-ci. Celles permettant plusieurs processeurs seront principalement observées, pour refléter les tendances actuelles d'accroissement du nombre de tâches à effectuer en parallèle.

Un premier exemple de ces configurations est d'ajouter un deuxième processeur PowerPC. Cet ajout apporte quelques perspectives intéressantes comme la possibilité d'un environnement multiprocesseur, ce qui apporte nombre de difficultés et de problèmes. Le support de plusieurs processeurs permet aussi d'accroître les capacités de calculs sans occasionner les problèmes de surchauffe et de consommation électrique associés

à l'augmentation des cadences d'horloge. Selon [10], 44% des projets de systèmes embarqués utilisaient, en 2006, plusieurs processeurs. Le fait que notre plateforme possède plusieurs processeurs permet donc de tester le support multiprocesseur des systèmes d'exploitation choisis et de notre flot. Une concurrence pour accéder aux ressources est par la même occasion créée. Plusieurs types d'interconnexions pourraient être utilisées, mais compte tenu du petit nombre de processeurs, un bus commun devrait être suffisant. En plus d'augmenter la capacité de calcul, la configuration que j'ai utilisée insère une méthode de déverminage que nous verrons en détail à la section 7.8.

Avec cette plateforme, nous pourrions aller encore plus loin dans le nombre de processeurs utilisés en synthétisant des microprocesseurs Microblaze ou tout autre microprocesseur que nous pourrions synthétiser sur les FPGA de Xilinx. Puisqu'une telle configuration nous permettrait d'avoir un grand nombre de processeurs, elle serait intéressante pour une application massivement parallèle.Nous pourrions même aller plus loin en utilisant plusieurs de ces cartes sur le même bus PCI et en communiquant via la mémoire de la machine hôte.

Une autre extension qui pourrait nous intéresser est de rajouter, sur le bus OPB, une partie de l'application que nous aurons synthétisée. Cela est nécessaire si nous voulons réaliser du codesign et tester notre conception, tout en étant relativement facilement réalisable.

La figure 3.1 montre un schéma de ces configurations.

CHAPITRE 4

SYSTÈME D'EXPLOITATION

4.1 Motivation

Bien que le but soit de supporter le plus de systèmes d'exploitation possible, il est impossible que ce que nous développons fonctionne sur chacun d'entre eux sans avoir à développer un module de compatibilité pour chacun d'eux. Il nous faudra donc limiter le nombre de systèmes d'exploitation supportés par notre flot tout en gardant en tête que nous voulons que le tout soit facilement portable vers de nouveaux systèmes d'exploitation.

J'ai donc sélectionné deux systèmes d'exploitation sur lesquels je vais effectuer le développement du flot. Ils sont suffisamment différents l'un de l'autre pour permettre d'apercevoir les difficultés qui pourraient surgir en tentant d'ajouter de nouveaux systèmes d'exploitation, mais assez semblables pour ne pas imposer de lourdes modifications pour passer de l'un à l'autre.

Dans cette section, je présenterai ces deux systèmes d'exploitation, les raisons qui m'ont poussé à les choisirs, les chaînes d'outils disponibles ainsi que les limitations que j'ai pu observer en utilisant ces systèmes d'exploitation sur la plateforme sélectionnée. Par la suite, je ferai un survol des autres systèmes d'exploitation embarqués.

4.2 OES/Mutek

4.2.1 Choix

Pour supporter l'application, nous avons décidé d'incorporer un système d'exploitation dans le flot de compilation. Il nous fallait donc choisir parmi les nombreux systèmes d'exploitation embarqués offerts. Les caractéristiques recherchées étaient la fiabilité et la malléabilité.

Nous avons opté pour un système d'exploitation développé dans le cadre du projet

Disydent[18] : Mutek. Ce dernier est léger, temps réel, multitâche et développé pour être facilement portable. Il supporte les modes multitraitement symétrique, multitraitement non symétrique avec ordonnanceur centralisé et multitraitement non symétrique avec ordonnanceur distribué. Nous aurions pu prendre d'autres systèmes d'exploitation embarqués comme eCos ou uCLinux, mais Mutek nous permet une plus grande latitude quant aux modifications du code et du fonctionnement. La probabilité que ces changements soient un jour pris en compte par ses créateurs semble aussi meilleure que dans un système d'exploitation plus avancé. Mutek implémente les fonctions de base de POSIX et est supporté par le simulateur CASS de Disydent.

4.2.2 Portage

Mutek était déjà porté sur plusieurs plateformes, mais il n'était pas porté sur le PowerPC. Nous avons donc été obligés de le porter sur la plateforme que nous avions choisie. Mutek est divisé en plusieurs parties plus ou moins indépendantes : libhandler, libc, libdpn et libpthread.

Libhandler est la partie qui contient, en théorie, toutes les fonctions qui dépendent du matériel. C'est cette partie qui doit être modifiée lorsqu'on veut porter le système d'exploitation sur une autre plateforme.

La portion libc de Mutek est une implémentation très réduite d'une bibliothèque standard C. Elle contient malheureusement une partie dépendante du matériel pour les entrées/sorties (*putchar* par exemple). Certaines études sont en cours pour améliorer le support des entrées/sorties et pour les rendre plus portables. J'ai donc modifié cette section pour supporter ma plateforme, mais en sacrifiant la compatibilité avec les autres plateformes pour lesquelles Mutek est déjà porté.

Libdpn est une librairie pour la communication entre les processus. Elle est entièrement portable et n'a nécessité aucune modification. Elle est basée sur le réseau de processus communicants de Kahn, mais nous pourrions y ajouter d'autres modèles d'exécution, tels que le passage de messages, les réseaux de Pétri ou des langages fonctionnels. La communication entre les processus se fait avec des FIFO.

Libpthread contient une implémentation des fils d'exécution POSIX. Elle contient

toutes les fonctions qui gèrent les processus, la synchronisation entre ceux-ci et la définition de l'ordonnanceur. Elle est donc, avec libhandler, le cœur du système d'exploitation. Elle n'est pas dépendante du matériel et j'ai donc eu à la modifier seulement pour effectuer du débogage.

Le portage nécessite donc seulement de modifier la librairie libhandler. Une autre partie importante, mais qui passe un peu sous silence est le script d'édition de liens. Cette partie est essentielle et dépend directement de la plateforme et est la seule qui nécessite d'être changée si l'on change la plateforme en gardant le même processeur.

4.2.3 Applications de test

Pour tester le système d'exploitation et le portage effectué de celui-ci, j'ai développé une série de tests unitaires afin de tester les différents services fournis par le système d'exploitation. Le premier de ces tests est un « *hello world* » standard afin de tester l'initialisation faite par le système d'exploitation et la communication par le port série. Le second est un petit programme multitâche afin de vérifier la gestion des fils d'exécutions et des primitives de synchronisation. Le troisième consiste en un producteur-consommateur simple afin de tester les communications.

Un autre de ces tests consiste en une série de modules s'exécutant dans des fils d'exécution différents et communiquant à l'aide de FIFO. Ce test contient cinq types de modules : un producteur qui génère des nombres aléatoires, quelques transmetteurs qui renvoient bêtement tout ce qu'ils reçoivent, quelques diviseurs qui envoient deux fois ce qu'ils reçoivent, quelques consommateurs qui font uniquement lire ce qu'ils reçoivent et finalement un comparateur qui prend deux entrées et qui les compare. Normalement, le comparateur devrait recevoir exactement la même donnée sur chacun de ses FIFO d'entrée. Chacun des modules a la possibilité d'afficher les valeurs lues ou écrites sur la console, ce qui permet une vérification manuelle des valeurs tout au long du test et de l'ordre de celles-ci.

Tous ces tests s'exécutent sans failles sur un ou deux PowerPC pendant de longues périodes de temps. Le plus gros de ces tests occupe moins de 7 Ko de mémoire, système d'exploitation compris.

Nous avions cependant besoin d'une application de plus grande amplitude pour tester véritablement les capacités multitâches de Mutek. Nous avons choisi d'utiliser l'application MJPEG pour tester les capacités de traitement et les différentes configurations matérielles possibles. L'application choisie nous permettra aussi de transférer certaines parties en matériel. La section 6 traite plus en profondeur de cette application.

4.2.4 Dna

Pendant la réalisation de mon projet de recherche, l'équipe qui développait Mutek a abandonné le projet pour un autre système d'exploitation : Dna[26].

Dna offre plus de fonctionnalités que Mutek. Notons, en autres, le support pour les entrées/sorties et le support de systèmes de fichiers (VFS). Le projet semble plus structuré et le développement est actif. Porter le travail que j'ai fait pour Mutek vers Dna pourrait donc être intéressant, mais nécessiterait un certain temps pour comprendre la structure du projet et savoir exactement ce qu'il faut modifier.

4.3 GNU/Linux

4.3.1 Choix

GNU/Linux est un système d'exploitation très connu et utilisé tant pour les serveurs, les stations de travail qu'au niveau embarqué. Son architecture ouverte laisse présager qu'il sera plus facile de le modifier et d'ajouter ce qu'il faut afin de pouvoir arriver à nos fins. De plus, la distribution ELDK (Embedded Linux Development Kit) version 3.0 vient déjà avec la carte Amirix.

GNU/Linux est souvent référé seulement comme étant Linux. Cependant, Linux ne réfère seulement qu'au noyau du système d'exploitation. GNU/Linux représente l'ensemble des logiciels utilisateurs et systèmes regroupés pour former un système d'exploitation complet.

4.3.2 Outils de développement

Après avoir essayé les outils disponibles avec la distribution ELDK fournie, j'ai rencontré un problème de librairies lorsque j'essayais de compiler de nouvelles applications. Je me suis donc tourné vers l'option d'une distribution maison. J'ai donc utilisé crosstool [16], un script pour compiler une chaîne d'outils GCC/Glibc en compilation croisée, et ptxdist[23], un autre script qui compile la distribution et des outils connexes.

Avec ces outils, j'ai compilé d'autres outils tels que Mono, un environnement d'exécution .net à code source ouvert. J'ai donc pu compiler et exécuter des modèles Esys.net, langage de simulation matériel développé à l'intérieur du laboratoire LASSO. Cette compilation ne s'est malheureusement pas faite sans heurts. Mono nécessite l'installation d'un grand nombre de librairies, par exemple glib, qui est une librairie de Gnome. Mono a aussi le désavantage d'utiliser les virgules flottantes, ce qui normalement ne devrait pas causer de problèmes puisque GCC inclut des routines d'émulation d'unité de virgule flottante, mais Mono fournit son propre compilateur et n'utilisait donc pas ces routines. J'ai donc dû aller modifier le noyau utilisé afin de lui faire capter et traiter les interruptions dues à l'utilisation du calcul à virgules flottantes. Le code existait pratiquement déjà dans le noyau, mais était désactivé, ce qui laisse présager une certaine instabilité ou du moins quelques problèmes avec le code qui était présent.

4.4 Limitations

Les deux systèmes d'exploitation utilisés ont leurs points forts et leurs points faibles. Mutek ne fournit à peu près aucune gestion des entrées et sorties et peu d'autres services que la gestion des processus et la communication. De plus, les programmes s'exécutent dans l'espace réel et sans restrictions (*supervisor mode*), ce qui compromet la sécurité du système si l'on est aux prises avec un programme malicieux ou mal conçu.

GNU/Linux fournit beaucoup plus de services en plus d'être plus répandu et développé, ce qui lui confère un sérieux avantage. Cependant, il est plus gros et complexe. Le modifier et le porter vers une nouvelle plateforme peut donc s'avérer beaucoup plus difficile. Il prend aussi plus d'espace et est légèrement plus lent, en raison de tous les

services qu'il doit fournir.

Les deux systèmes d'exploitation ont certains points faibles en commun, mais qui sont surtout imposés par la plateforme. Premièrement, le support de l'unité de calcul à virgules flottantes n'est pas présent sur le processeur. Le compilateur GCC fournit une émulation de celles-ci et ne les utilise donc pas. Cependant, GCC n'est pas le seul compilateur utilisable et une interruption permet de capter les tentatives d'utilisations des virgules flottantes et de les traiter de manière transparente à l'application. Aucun des deux systèmes d'exploitation n'utilise cette option, mais l'ajouter est simple, surtout dans le cas de GNU/Linux.

Une autre limitation importante est que les deux systèmes d'exploitation éprouvent des difficultés à s'exécuter sur les deux processeurs disponibles. Mutek arrive parfois dans un état où tous ses processus sont en attente d'un autre (interblocage) et GNU/Linux se met à s'exécuter beaucoup plus lentement qu'avec un seul processeur. Les tests unitaires n'exhibent pas le problème d'interblocage éprouvé avec MJPEG sur Mutek.

4.5 Autres systèmes d'exploitation disponibles

La présente section présente une panoplie d'autres systèmes d'exploitation que nous aurions aussi pu utiliser. Les analyses reposent sur les informations disponibles sur ces systèmes d'exploitation selon différentes sources disponibles comme, par exemple, le site du fabricant. Le grand nombre de systèmes d'exploitation disponibles nous empêche d'en faire une étude exhaustive.

Un sondage du magazine *Embedded Systems Design* sur les tendances dans les systèmes embarqués [10] porte sur plusieurs facettes de la conception de systèmes embarqués. La section sur les systèmes d'exploitation est celle qui nous intéresse. Elle est résumée et expliquée dans [27]. Une des données qui nous intéresse particulièrement est les critères d'évaluation d'un système d'exploitation. Il ressort que les principaux critères sont, dans l'ordre : les performances temps réel, les processeurs supportés, la chaîne d'outils et les royautés. Je vais donc essayer de comparer ces différents facteurs.

Une autre donnée intéressante est la liste des systèmes d'exploitation les plus po-

pulaires. Selon cette étude, ce serait, dans l'ordre : VxWorks, XP Embedded, Windows CE, DSP/BIOS, Red Hat Linux, QNX, RTX, μC/OS, Nucleus et Integrity. Thread X est séparé en deux parties (selon le distributeur) mais aurait été classé parmi les précédents. J'ai donc inclus ces systèmes d'exploitation ainsi que certains autres qui me semblent prometteurs dans la liste qui suit. Une autre donnée qui aurait pu être intéressante est l'empreinte mémoire de chaque système d'exploitation. Cependant, cette information dépend dans bien des cas de la configuration choisie, du processeur utilisé et même parfois de l'application que l'on désire exécuter. Pour que cette donnée soit utile, il faudrait savoir avec quel ensemble de services la taille minimale de chaque système d'exploitation est calculée ; information qui est rarement disponible.

4.5.1 Temps réel strict ou mou

L'attribut temps réel d'un système d'exploitation spécifie si ce dernier fournit des primitives afin de fixer l'échéance d'une tâche. Une attention particulière doit donc être portée à l'ordonnanceur, aux interruptions et aux changements de contexte. Si le système d'exploitation veut s'afficher comme étant temps réel strict, il doit garantir que chaque opération, ou le sous-ensemble compatible temps réel de celles-ci, s'exécute dans un temps borné. Le comportement du système est donc garanti d'être déterministe, même dans des conditions de charge intense. Le temps réel mou fournit les primitives pour fixer l'échéance, mais celle-ci peut être dépassée de temps à autre. Dans certaines applications, par exemple dans les domaines de l'automobile, de l'aéronautique et des centrales nucléaires, un dépassement du délai pour une tâche ou l'abandon de celle-ci peut avoir des conséquences catastrophiques. Ces applications nécessitent un temps réel strict. D'autres applications, notamment dans le traitement de signal et d'image, peuvent souvent voir quelques opérations échouer sans qu'il y ait de conséquences dramatiques. Une trame de DVD non affiché peut être désagréable, mais a des conséquences bien moindres qu'un avion qui s'écrase. La certification DO-178B existe afin de garantir le temps de réponse des applications dans le domaine aéronautique, mais coûte cher. Tous les systèmes d'exploitation assurant être temps réel prétendent aussi être temps réel strict, mais certains sont aussi certifiables DO-178B. Ils ont habituellement une version -178B.

4.5.2 DSP/BIOS

DSP/BIOS est un système d'exploitation conçu par Texas Instruments pour les systèmes DSP. Il utilise des API non standardisés et des outils de développement propres à TI. Il est temps réel strict, supporte principalement le langage C. DSP/BIOS est libre de redevances à l'exécution. Malgré que le seul processeur supporté soit l'ARM et disponible uniquement pour quelques plateformes.

4.5.3 eCos

eCos est un système d'exploitation libre et à code source ouvert, sans redevances et supportant le temps réel strict. Il supporte les processeurs ARM, Hitachi H8300, Intel x86, MIPS, Matsushita AM3x, Motorola 68k, PowerPC, SuperH, SPARC et NEC V8xx. La version 405 du PowerPC, soit celle sur les Virtex 2 Pro de Xilinx, et le Nios II d'Altera sont supportés . Il supporte les API standardisées POSIX et μITRON. Il utilise principalement le langage C et les outils de développement de GNU. Le support des systèmes à processeurs multiples est limité. Si la préemption et le multitâche ne sont pas nécessaires, eCos donne la possibilité d'être compilé sans le noyau, ce qui revient à utiliser une librairie. eCos est le système d'exploitation utilisé par la Playstation 3 de Sony, les radios satellites Sirius et dans certaines télévisions haute définition LCD de Samsung.

4.5.4 Integrity RTOS

Integrity vient en plusieurs versions : Integrity, Integrity-178B (pour les applications critiques), Integrity PC (stations de travail), velOSity (noyau de Integrity), μ-velOSity (micronoyau). Les informations qui suivent sont pour Integrity, version de base.

Integrity est un système d'exploitation temps réel strict pour processeur 32 et 64 bits avec unité de gestion de mémoire développée par la compagnie Green Hills Software. Il est propriétaire, mais sans redevances, et utilise des outils propriétaires pour la compilation. Le paradigme objet ainsi que les langages C, C++, C++ embarqué et Ada sont privilégiés. Les processeurs disponibles sont ARM, Blackfin (Analog Devices), ColdFire

(Freescale), MIPS, PowerPC (IBM et Freescale) et XScale. Les API POSIX et μITRON sont supportées et celui de VxWorks l'est partiellement. Intergity supporte les systèmes multiprocesseurs distribués. Integrity est en autre utilisé dans les avions de combat F-16 et F-35.

4.5.5 LynxOS

LynxOS est disponible, lui aussi sous différentes versions soient : LynxOS (pour systèmes embarqués), LynxOS-SE (stations de travail) et LynxOS-178 (applications critiques). LynuxWorks fournit aussi un hyperviseur, LynxSecure et une distribution GNU/-Linux, BlueCat. La version qui nous intéresse est celle pour systèmes embarqués, soit LynxOS.

LynxOS est développé par la compagnie LynuxWorks. Contrairement à ce que son nom peut évoquer, LynxOS n'est pas une distribution GNU/Linux. Cependant, Lynux-Works affirme que l'interface binaire de ses applications est compatible à celui de GNU/-Linux, ce qui signifie que les applications compilées pour GNU/Linux devraient fonctionner sous LynxOS. LynxOs devrait donc supporter l'API POSIX aussi bien que GNU/-Linux. Le développement d'application se fait en langage C, C++ ou java à l'aide d'un plugiciel pour Eclipse, Luminosity. La suite GNU peut aussi être utilisée. LynxOS supporte les systèmes multiprocesseurs symétriques, utilise l'unité de gestion de mémoire et supporte les contraintes temps réel strictes. Les processeurs supportés sont ARM, PowerPC, MIPS et x86. LynxOS est utilisé dans le simulateur de vol de l'Airbus A380, le système de navigation satellite Galileo et dans le système de contrôle aérien ARTS.

4.5.6 Nucleus RTOS

Nucleus est un système d'exploitation temps réel strict développé par Mentor Graphics. Les langages principalement supportés sont C et C++ avec les API POSIX et μITRON. Nucleus comporte une extension pour le support d'unité de gestion de mémoire et supporte une multitude de processeurs, entres-autres ARM, Atmel, Cell, FreeScale Coldfire, M Core, Microblaze, MIPS, NIOS (I et II), PowerPC, Tricore, x86, Xscale

et Xtensa. La suite EDGE fournit les outils de développement. Nucleus supporte les environnements multiprocesseurs symétriques ou non. Nucleus est utilisé par Honeywell dans son "Critical Terrain Awareness Technology", dans un détecteur à rayon X de Anrad et dans le défibrillateur de ZOLL.

4.5.7 OSE

Encore une fois, OSE (Operating System Embedded) est offert en plusieurs versions, soit OSE, OSEck (pour DSP) et OSE Epsilon (minimaliste).

OSE est un système d'exploitation temps réel strict développé par la compagnie ENEA. La communication interprocessus se fait par passage de message à haut niveau et est supportée par un API propre à OSE. L'utilisation d'unité de gestion de la mémoire est optionnelle pour les familles de processeurs supportés, soient ARM, IXP2400, MIPS 32, PowerPC et OMAP. OSE supporte les multiprocesseurs distribués. Le développement d'applications se fait dans un plugiciel pour Eclipse propre à OSE et utilise les outils GNU associés surtout aux langages C et C++. OSE est utilisé dans le traitement des gaz de combustion de la compagnie ABB, un système d'imagerie infrarouge de Agema et dans le système de contrôle de soudure de Aro Controls.

4.5.8 Plan9 (Inferno)

Plan9 se veut un système d'exploitation de recherche différent de ce qui existait avant, bien qu'il utilise, ou plutôt approfondi, certains concepts d'UNIX. Inferno est basé sur Plan9, mais orienté vers le marché des systèmes embarqués. Le développement d'applications se fait en LIMBO, un langage modulaire avec une syntaxe semblable à C et compilé vers un pseudo-code binaire indépendant du matériel. Les pilotes de matériels sont, quant à eux, écrits en C. L'architecture de Inferno permet d'utiliser des ressources distantes sans que le programme ne s'en rende compte. Inferno peut s'exécuter sur les processeurs x86, Xscale, PowerPC, StrongARM et SPARC. De plus, Inferno peut s'exécuter en tant qu'application sur certains systèmes d'exploitation. Inferno est disponible sous licence libre.

4.5.9 QNX neutrino RTOS

Neutrino RTOS est un système d'exploitation temps réel strict développé par QNX. QNX est un système à micronoyau et donc tous les processus qui peuvent être exécutés en mode protégé le sont. Les pilotes matériels ont donc accès à un espace d'adresse virtuel et ne peuvent communiquer avec le matériel sans passer par le système d'exploitation. Presque n'importe quelle partie du système d'exploitation, peut, en cas de défectuosité, être redémarrée sans affecter l'intégrité du système. Cela signifie cependant qu'une unité de gestion de la mémoire est indispensable. Neutrino supporte l'API POSIX et utilise un environnement propriétaire, Momentics, avec les langages C et C++. Les processeurs ARM, MIPS, PowerPC, SH, et x86 sont supportés. QNX est utilisé par le robot sous-marin autonome Celtus II de Lockheed/Martin et par le système de correction de la vue par le laser de IntraLase.

4.5.10 RTEMS

RTEMS (Real-Time Executive for Multiprocessor Systems) est un système d'exploitation temps réel strict libre et à code source ouvert. RTEMS supporte les API POSIX, RTEID/ORKID et μITRON ainsi que les langages C, C++ et ada. Le développement se fait avec la chaîne d'outils GNU. Il supporte les systèmes multiprocesseurs homogènes et hétérogènes. Les processeurs supportés sont ARM, Blackfin, H8300, m68k, MIPS, NIOS II, or32, PowerPC, SuperH, Sparc, Ti/C4x et x86. RTEMS ne supporte pas les unités de gestion de mémoire. RTEMS est utilisé dans la sonde Venus Express et dans la tondeuse autoguidée HybridZ.

4.5.11 ThreadX

ThreadX est un système d'exploitation temps réel strict développé par Express Logic et distribué sur un modèle sans redevances et où le code source est disponible, mais n'est pas libre. ThreadX supporte les unités de gestion de mémoire ainsi qu'une vaste gamme de processeurs dont : ARM, Blackfin, ColdFire, M-CORE, Microblaze, MIPS, Nios II, PowerPC, x86, XScale et Xtensa. Le développement peut se faire sur plusieurs envi-

ronnements comme le Workbench de Wind River, Eclipse, MULTI de GHS ou même la suite GNU pour certaines plateformes. ThreadX supporte les API OSEK, POSIX et μITRON. ThreadX se présente comme une librairie C, qui est donc avantagé comme langage de développement. ThreadX supporte les environnements multiprocesseurs symétriques ou asymétriques. ThreadX est utilisé dans les puces bluetooth pour cellulaire et PDAs de Broadcom, la sonde Deep Impact de la NASA et le moniteur médical sans fil de Welch-Allyn Protocol.

4.5.12 μc/OS-II

μc/OS-II est un système d'exploitation temps réel strict développé par Micrium et disponible avec accès au code source et sans redevances. μc/OS-II est disponible pour bon nombre de processeurs, dont ARM, Blackfin, ColdFire, Microblaze, M-CORE, Nios (I et II), PowerPC, x86 et XScale. μc/OS-II offre un support optionnel pour les unités de gestion de mémoire. Il supporte l'API OSEK, l'environnement de développement suggéré pour μc/OS-II diffère selon le processeur. Par exemple, μc/OS-II utilise l'EDK de Xilinx pour le développement sur PowerPC 405. Le principal langage supporté est C. μc/OS-II est utilisé dans le système de vol Quest ! xc de Active Flight Systems et le système de surveillance sismique pour barrage et centrales nucléaires NCC de Syscom.

4.5.13 VxWorks

VxWorks est un système d'exploitation temps réel strict propriétaire développé par Wind River. Il supporte les systèmes multiprocesseurs symétriques et asymétriques et utilise l'unité de gestion de mémoire. Il supporte l'API POSIX ainsi que les langages C et C++. Les applications peuvent être réalisées avec la suite GNU ou avec des outils propriétaires. Les processeurs supportés sont ARM, ColdFire, Intel, MIPS, PowerPC, SuperH et Xscale. VxWorks est notamment utilisé dans les missions spatiales Mars Pathfinder, Stardust, Spirit et Opportunity.

4.5.14 Windows CE

Windows CE est un système d'exploitation propriétaire développé par Microsoft. Il utilise l'unité de gestion de mémoire et supporte les familles de processeurs ARM, MIPS, SH4 et x86. Les applications sont construites à l'aide d'un plugiciel pour Visual Studio de Microsoft et les langages supportés sont C, C++, C# ou VB (.NET Compact Framework)). Windows CE est utilisé dans l'équipement de surveillance médicale à domicile de Zoe Medical et dans les systèmes de navigation personnels de Mobile Crossing.

4.6 Comparaison

La figure 4.1 fait un résumé des possibilités des différents systèmes d'exploitation répertoriés précédemment.

Tableau 4.1: Comparaison des systèmes d'exploitation étudiés

Nom	Temps Réel	Processeurs supportés				Chaîne d'outils		Royautés	API
		Arm	Mips	PowerPC	X86	GNU/GCC	Autres		
DSP/BIOS	Oui	X					X		
eCos	Oui	X	X	X		X			I+P
Integrity	Oui	X	X	X			X		I+P+V
GNU/Linux	Non	X	X	X	X	X			P
LynxOS	Oui	X	X	X	X	X	X	X	P
Mutek	Oui	X	X	X		X			P
Nucleus	Oui	X	X	X	X		X	X	P+I
OSE	Oui	X	X	X		X	X		
Plan9	Non	X		X	X		X		
QNX	Oui	X	X	X	X		X	X	P
RTEMS	Oui		X	X	X	X			I+P
ThreadX	Oui	X	X	X	X		X		I+O+P
μC/OS-II	Oui	X		X	X		X		O
VxWorks	Oui	X	X	X	X	X	X	X	P+V
WindowsCE	Oui	X	X		X		X	X	

API : I = μITRON, O = OSEK, P = POSIX, V= VxWorks.
Certains systèmes d'exploitation supportent d'autres API, p. ex. : Windows et l'API .Net
Il existe des programmes superviseurs pour ajouter une couche temps réel à GNU/Linux, p. ex. : Xenomai

4.6.1 Conclusion

Nous pouvons donc constater qu'il existe une bonne variété de systèmes d'exploitation embarqués. Nous ne pouvons malheureusement pas tous les supporter et nous en avons, par conséquent, sélectionné deux. Ces deux systèmes d'exploitation offrent une bonne différence de taille et de fonctionnalités. Ils permettront donc de bien tester la portabilité du flot entre différents systèmes d'exploitation. Si nous parvenons à faire un flot permettant de supporter ces deux systèmes d'exploitation, il est fort probable que nous puissions intégrer n'importe quel autre par la suite. Les deux systèmes d'exploitation sélectionnés utilisent l'API POSIX, ce qui nous simplifie grandement la tâche.

CHAPITRE 5

LANGAGE

5.1 Motivation

Peu importe où nous voulons aller, nous avons besoin d'un point de départ. Dans le cas qui nous intéresse, le point de départ va être un code écrit dans un langage de programmation, mais lequel ?

Le développement des langages de programmation modernes fait en sorte qu'il y a maintenant une panoplie de choix offerts. Trouver le langage le plus adapté à ses besoins peut donc être assez compliqué.

Nous cherchons un langage avec lequel nous pourrons éventuellement décrire un système logiciel/matériel. Ce langage devra donc être suffisamment de haut niveau pour ne pas trop rebuter les programmeurs, mais doit être suffisamment simple pour être éventuellement synthétisé en matériel.

Si l'on choisit un langage de haut niveau, les abstractions seront plus faciles à faire et l'on disposera probablement d'outils plus élaborés pour la conception et le déverminage, certains permettant même de faire de l'introspection et permettant de modifier du code aisément. Cependant, avec la réalité des systèmes embarqués, où les capacités sont limitées et où la communication avec des composantes matérielles spécialisées est fréquente, un langage de bas niveau est peut-être préférable.

Il faut donc, dans un premier lieu, choisir dans une liste de langages potentiels celui qui nous semble le plus approprié. Peu importe le langage choisi, il faudra aussi vraisemblablement limiter la portion du langage supporté et probablement suggérer quelques additions ou fonctions nécessaires à ce que nous voulons accomplir. Il faudra finalement trouver la chaîne d'outils pour ce langage fonctionnant avec les systèmes d'exploitation choisis.

5.2 C

5.2.1 Avantages et inconvénients

C est un langage d'assez bas niveau pour pouvoir dialoguer directement avec le matériel et est parfois le seul langage officiellement supporté par certains systèmes d'exploitation. Les API sont parfois différentes d'un système d'exploitation à un autre, mais les API POSIX et μITRON sont supportées par beaucoup de systèmes d'exploitation embarqués. C est le langage officiellement supporté par le plus de systèmes d'exploitation et semble donc le choix idéal pour assurer un maximum de portabilité.

Cependant, le bas niveau de C est également une faiblesse. Le temps de développement se trouve habituellement augmenté par rapport à un langage de plus haut niveau et les erreurs sont plus difficilement détectées et parfois très subtiles. La gestion manuelle de la mémoire peut occasionner des fuites de celle-ci et mener à une panne généralisée du système.

Un autre problème avec ce langage est de déterminer les endroits où nous pouvons faire le partitionnement logiciel/matériel. Les appels de fonctions semblent être un endroit intéressant pour effectuer les coupures, mais il faut s'assurer qu'aucune variable globale n'est utilisée à l'intérieur de celles-ci. Il faut également recréer un protocole d'appel qui doit, si l'on veut rendre la conception indépendante du partitionnement, être bloquant.

5.2.2 Limitations et ajouts

Les problèmes induits par l'utilisation de C comme langage de source nous obligent donc à faire certains choix. Nous devons, dans un premier temps, nous limiter à supporter un seul API. Le choix de l'API va décider, avec le modèle de pilote de périphérique, quel système d'exploitation nous pouvons supporter. Puisque les systèmes d'exploitation que nous avons choisis utilisent tous les deux l'API POSIX, nous allons donc nous restreindre à cet API.

La plupart des processeurs embarqués n'ayant pas d'unité d'arithmétique à virgule flottante, il serait sage de limiter les programmes à l'utilisation de nombres entiers.

Cependant, certains algorithmes de traitement de signaux sont plus simples avec des nombres à virgule flottante. Heureusement, certains compilateurs, dont GCC, transforment automatiquement les opérations à virgules flottantes présentes dans le code pour qu'elles utilisent une librairie logicielle. Le PowerPC présent sur notre plateforme offre aussi une interruption lorsqu'il croise une telle opération, permettant donc de brancher à une routine exécutant l'opération demandée. Linux offre un support pour cette interruption, moyennant un peu de bidouillage dans le noyau fourni avec la plateforme.

Le risque posé par l'allocation manuelle de la mémoire peut être contourné en n'offrant pas de fonctions d'allocation dynamique de mémoire, ce qui est plutôt contraignant, ou en établissant une règle, lors de l'élaboration des programmes, que la mémoire nécessaire devrait être réclamée au début de l'exécution du programme. De cette façon, nous évitons de mauvaises surprises lorsque l'application fonctionne pendant un certain temps avant de planter.

Reste le problème de partitionnement logiciel/matériel. Pour contourner ce problème, j'ai restreint les programmes supportés à ceux qui utilisent le modèle d'exécution des réseaux de processus de Kahn. Ce modèle consiste en plusieurs processus communiquant entre eux à l'aide de files de type premier entré, premier sorti de taille infinie. La réalité des systèmes embarqués nous oblige bien sûr à mettre une limite sur la taille des files. Nous utiliserons la librairie DPN (dysident process network) fournie avec les sources de Mutek, aussi disponible pour GNU/Linux et probablement facilement portable vers n'importe quel système d'exploitation supportant POSIX.

5.2.3 Chaîne d'outils

La plupart des systèmes d'exploitation recensés au chapitre 4 offrent une chaîne d'outils permettant l'utilisation du langage C. La chaîne d'outils n'est donc pas un problème pour C.

5.2.4 Synthèse matérielle

Bien que les programmeurs considèrent en général C comme étant de bas niveau, les concepteurs de matériel utilisent des langages de plus bas niveau que C et considèrent donc celui-ci comme de haut niveau. Cependant, dû à la popularité du langage C auprès des programmeurs, la synthèse comportementale de C vers des langages RTL a souvent été étudiée. La plupart des approches ajoutent des macros à la librairie standard de C afin de fournir des primitives pour la description du matériel. Dans cette catégorie d'outils, nous trouvons, entre autres, Handel-C, FpgaC et Impulse C. La variété d'outils nous laisse présager que nous pourrons facilement en trouver un qui pourrait transformer les parties désignées pour le matériel en langage de description matériel. Vu que la plupart des outils ne prennent qu'un sous-ensemble de C, il est possible que quelques ajustements soient nécessaires lors de la synthèse du système. Puisque cela dépasse l'étendue désirée de ce mémoire, je laisse la démonstration de l'arrimage d'un tel outil à l'architecture présente pour de futurs travaux.

5.3 C# et ESys.net

5.3.1 Avantages et inconvénients

C# est un langage de haut niveau permettant de l'introspection et est normalement compilé vers un pseudo-code binaire. ESys.net[11] est un langage de description de systèmes basé sur C#.

L'ensemble du langage C# nécessite l'emploi d'une bibliothèque assez volumineuse et n'est donc pas souhaitable pour la plupart des plateformes embarquées. Cependant, nous pourrions nous limiter à un sous-ensemble comme Esys.net.

L'emploi de ce langage nous obligerait à nous limiter aux systèmes d'exploitation qui sont assez complets pour supporter une implémentation C# de base ou à développer une traduction de C# vers un langage plus largement supporté, par exemple C ou C++. Le développement d'une telle traduction nécessiterait une bonne quantité de travail. Heureusement, il existe déjà quelques implémentations qui existent sur le marché.

L'avantage d'utiliser ce langage est qu'il fournit de fortes capacités d'introspection ainsi qu'une simulation du système qui pourrait être plus complète et plus rapide. Le plus haut niveau de ce langage rendrait aussi possible le développement d'une architecture de cosimulation utilisant une partie matérielle déjà synthétisée. Celle-ci pourrait communiquer avec un simulateur sur une machine hôte.

5.3.2 Limitations et ajouts

Esys.net définit déjà quelques limitations et ajouts à la norme C#. Bien que du code Esys.net strict ne soit probablement pas avantageux pour écrire la partie logicielle, nous pourrions quand même utiliser un code écrit complètement dans ce langage afin d'avoir une implémentation qui fonctionne et que nous pourrions par la suite l'agrémenter pour qu'elle supporte une plus grande partie de C#. Le code Esys.net est loin d'être idéal pour la partie logicielle puisqu'elle contient des composants de bas niveau qui sont utiles pour la synthèse, mais qui ralentissent le code si on les conserve en logiciel.

Nous pourrions réutiliser les idées utilisées pour le langage C et limiter les jonctions entre le matériel et le logiciel à des communications via une interface de communication de type FIFO.

5.3.3 Chaîne d'outils

La bibliothèque d'exécution nécessaire pour l'exécution de code C# peut s'avérer assez volumineuse. Même si nous trouvions un outil pour transférer de C# vers C++, le code généré risque d'avoir besoin d'un nombre substantiel de fonctions. De plus, ce ne sont pas tous les systèmes d'exploitation qui supportent le C++, ce qui aurait pour effet de nous bloquer beaucoup de choix.

5.3.4 Synthèse matérielle

Actuellement la synthèse de ESys.net vers le matériel doit être faite manuellement, mais il est envisageable que quelqu'un développe les outils pour faire la synthèse com-

portementale de ce langage. L'utilité d'ESys.net se résume donc présentement à la simulation.

5.4 Scheme

5.4.1 Avantages et inconvénients

Scheme est un langage fonctionnel minimaliste. J'ai choisi d'étudier la possibilité d'utiliser Scheme comme langage source à la suite d'articles sur la synthèse matérielle à partir de Scheme[5]. Le paradigme de la programmation fonctionnelle est souvent utilisé pour la vérification à haut niveau, mais très peu pour la synthèse.

5.4.2 Limitations et ajouts

Scheme étant déjà minimaliste, peu de limitations doivent être faites, surtout du côté logiciel. La scission entre le logiciel et le matériel peut se faire lors de l'appel d'une fonction.

Contrairement à d'autres langages comme C et C#, nous pouvons savoir, à certaines étapes de la compilation, quelles seront les variables qui seront utilisées plus tard. Avec l'aide des continuations, nous pouvons aussi savoir quelle fonction nous devons appeler en recevant les résultats du matériel. Puisque nous savons combien de valeurs doivent être envoyées à notre module matériel et combien on doit en lire, nous pouvons envoyer ces valeurs par le moyen d'un FIFO. Malheureusement, il faut entrer dans le code d'un compilateur pour avoir accès à ces informations, ce qui peut s'avérer assez fastidieux.

Le standard Scheme ne prévoit pas de primitives pour gérer la concurrence mais, tout comme le C, certaines librairies et implémentations proposent des primitives. Il est nécessaire d'avoir de la concurrence pour éviter que les entrées/sorties vers le matériel ne retardent trop le logiciel. En ayant de la concurrence, le processeur peut effectuer d'autres tâches en attendant que les tâches matérielles s'effectuent. Avoir de la concurrence dans le code permet aussi de mieux profiter de la parallélisation que peut procurer le matériel si l'on décide de synthétiser des parties du système.

La gestion automatique de la mémoire de Scheme peut entraîner de sérieux maux de tête dans un environnement embarqué. Nous ne voulons pas perdre beaucoup d'espace mémoire et nous ne voulons pas non plus que notre système ait de pauses prolongées. Il faut donc que le glaneur de cellules soit très performant tant pour l'espace mémoire utilisé que pour la latence qu'il peut introduire.

5.4.3 Chaîne d'outils

Plusieurs compilateurs Scheme fournissent la possibilité d'avoir une sortie en langage C. La portabilité n'est donc pas un problème, sauf pour les systèmes d'exploitation minimalistes qui ne fournissent pas une bibliothèque C complète. Il faudrait modifier un des compilateurs offerts afin d'intégrer les modifications nécessaires à la synthèse des communications. Cependant, une bonne partie du code qui serait nécessaire aux communications avec le langage C serait aussi nécessaire avec le langage Scheme.

5.4.4 Synthèse matérielle

Comme j'ai mentionné en 5.4.1, quelques études ont été faites sur la synthèse à partir de Scheme. Les travaux d'Étienne Bergeron[6] vont même un peu plus loin que la synthèse comportementale en proposant une méthode de synthèse dynamique.

5.5 Communications

La décision de limiter les programmes à un modèle de processus communicants nous simplifie grandement la tâche pour ce qui est des communications. Grâce à l'environnement Dysident avec lequel Mutek vient, nous avons déjà une bibliothèque implémentant des FIFO logiciels. Cette bibliothèque, DPN, va prendre en charge les échanges entre logiciels.

La communication entre matériels peut être implémentée par des FIFO matériels.

Les communications entre logiciel et matériel et de matériel à logiciel demandent un peu plus de réflexion. De plus, l'implémentation va largement dépendre du système

d'exploitation sous-jacent ainsi que de la plateforme. Chaque système d'exploitation a sa façon d'accéder au matériel.

Plusieurs, comme Linux, utilisent des pilotes de périphériques pour s'occuper de cette tâche. Certaines initiatives, notamment celle du *Uniform Driver Interface* (UDI) ou interface de pilote unifiée, ont essayées d'établir une interface standard entre les différents systèmes d'exploitation, mais n'ont pas réussi. Il faut donc développer un pilote pour chaque système d'exploitation que nous voudrions supporter.

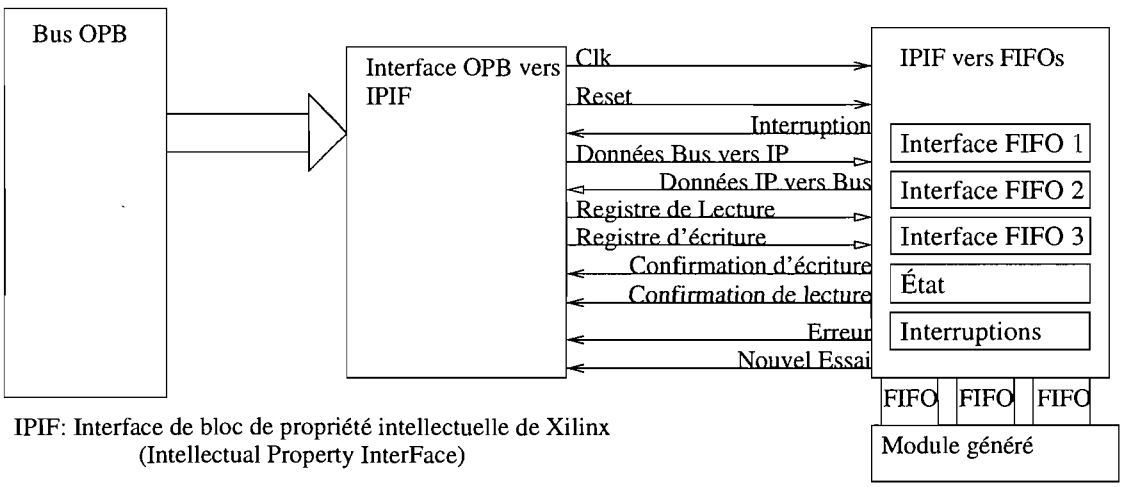
L'initiative de l'*Uniform Driver Interface* propose aux pilotes de périphériques une interface standardisée afin d'accéder à leur environnement. Avec l'UDI, les appels au système d'exploitation et au matériel se font de la même façon sur toutes les plateformes supportant l'UDI. Cela permettrait aux pilotes de périphériques de facilement supporter une vaste sélection de systèmes d'exploitation avec le même code. L'initiative est malheureusement demeurée au stade de preuve de concept.

Certains systèmes d'exploitation, surtout les plus minimalistes, donnent la possibilité aux logiciels utilisateurs d'accéder directement au matériel. L'écriture du pilote se trouve donc simplifiée, au détriment d'une sécurité d'exécution et d'une facilité de déverminage.

J'ai choisi d'implémenter la communication de logiciel à matériel en utilisant un registre projeté en mémoire pour chaque FIFO et un registre (ou plusieurs s'il y a trop de FIFO) de statut. L'écriture dans un FIFO se fait en lisant le registre de statut afin de s'assurer que le FIFO n'est pas plein et en écrivant, le cas échéant, les données à l'adresse mémoire appropriée. La lecture se fait avec un procédé similaire. La figure 5.1 illustre le module matériel généré pour les communications logiciel vers matériel et matériel vers logiciel. Dans le cas d'indisponibilité du FIFO, le processeur passe le contrôle à un autre processus et vérifie de temps à autre si le statut n'a pas changé. Cette méthode est loin d'être efficace et devrait être modifiée pour laisser place à une alternative basée sur les interruptions.

Du côté logiciel, l'appel au pilote peut être fait à l'intérieur du code de FIFO de la librairie DPN, mais cela nécessite de retirer toutes références à la fonction définissant le module. L'approche que j'ai développée est plutôt de remplacer le code du module

Figure 5.1: Structure du module matériel



IPIF: Interface de bloc de propriété intellectuelle de Xilinx
(Intellectual Property InterFace)

par un code qui fait l'interface entre les FIFO entre logiciels de DPN et le module matériel. Puisque nous ignorons l'ordre et la quantité de données à lire ou écrire sur chaque FIFO, un seul fil d'exécution essayant une lecture à tour de rôle sur chacune d'elle risquerait fort de bloquer alors que certaines opérations sont encore possibles. Il faudrait donc un système de FIFO possédant des lectures et écritures non blocantes ou un moyen de vérifier l'état du tampon. La librairie DPN ne fournissant pas ces primitives, une attention particulière doit être donnée aux problèmes d'interblocages. Pour m'assurer qu'un tel problème n'intervient pas, j'ai placé l'interface pour chaque FIFO dans un fil d'exécution indépendant. Nous devons aussi nous assurer que les FIFO qui sont traités uniquement en matériel ne sont pas accédés.

5.6 Conclusion

Je n'ai pas fait une revue de tous les langages disponibles, mais une constante ressort cependant du lot : passer par un langage de bas niveau qui puisse accéder aux ressources matérielles est nécessaire pour la poursuite de mon travail. Je vais donc utiliser un langage de bas niveau pour la suite de mon travail. Il pourrait cependant être intéressant de s'attarder à d'autres langages de plus haut niveau par la suite, en se basant sur la métho-

dologie développée pour le langage C. Il semble cependant peu utile de débuter par un langage de haut niveau puisque cela nécessiterait d'abord d'effectuer le travail pour un langage de bas niveau, fort probablement C, en plus du travail nécessaire pour le langage de haut niveau. Afin de limiter la charge de travail et le temps nécessaire à l'élaboration de mon flot, je me limiterais donc au langage C.

Dans les langages itératifs, la limitation au modèle de processus communicants nous simplifie la tâche, mais demande quelques réflexions et du travail d'implémentation. Dans les langages fonctionnels, nous pouvons, à certaines étapes de la compilation, faire une analyse afin de découpler une partie du code pour le transformer en matériel. Nous pouvons ensuite abstraire ces bouts de code en processus communicants avec des FIFO pour passer les arguments et variables au matériel et pour les récupérer une fois le travail terminé.

CHAPITRE 6

APPLICATION DE RÉFÉRENCE

6.1 Motivation

Pour tester mon raisonnement et identifier les failles potentielles dans celui-ci, je me suis servi d'une application parallélisée et qui nécessite une capacité de calcul assez grande pour justifier de mettre une partie de celle-ci en matériel. J'ai donc utilisé l'application MJPEG qui consiste à décoder plusieurs images JPEG concaténées dans le même fichier JFIF.

L'application est parallélisée et contient certaines fonctions nécessitant une bonne charge de calcul, donc ayant avantage à être transformées en partie en matériel. L'application initiale a besoin d'un fichier d'entrée et produit une sortie par un FIFO, mais elle peut être modifiée afin de prendre son entrée en mémoire et valider sa sortie avec celle produite par une exécution de référence. L'application peut donc tourner entièrement en mémoire, ce qui est utile si l'on veut utiliser un système d'exploitation minimaliste comme Mutek.

Je vais donc, dans ce chapitre, présenter l'application MJPEG que j'ai utilisée pour m'aider à détecter les difficultés et tester les différentes parties. Je vais aussi présenter certains résultats que j'ai obtenus.

6.2 Implémentation logicielle

L'application contient plusieurs processus communiquant à l'aide de FIFO : un générateur de trafic (TG) qui lit les octets d'un fichier ou de la mémoire, un démultiplexeur (Demux) qui décode les informations du TG et qui les envoie au module désigné, un module qui calcule des coefficients pour la transformation de cosinus discret, *Discrete cosine transform* ou DCT, (*Variable lenght decoder*, VLD), un déquantificateur DCT (*Inverse Quantization*, IQ), un réordonnanceur de bloc DCT (*ZigZag*, ZZ), une transformation DCT inverse (*Inverse Discrete Cosine Transform*, IDCT), un constructeur

de ligne (libu) et un afficheur (*Random Access Memory Digital-to-Analog Converter*, RAMDAC). Demux s'occupe de découper le fichier d'entrée en différents blocs, telles les tables de Huffman, tables de quantification et données d'image. VLD s'occupe de décoder les données du fichier qui ont été codées à l'aide d'un algorithme de Huffman, d'où la nécessité que les tables de Huffman lui soient transmises. IQ s'occupe de faire une multiplication élément à élément des tables de quantification et des données d'entrée. ZZ s'occupe de réordonnancer les éléments à l'entrée ; cette étape est utilisée pour augmenter l'entropie et favoriser la compression. IDCT est une transformée de cosinus discrète inverse, ce qui a pour effet de passer du domaine de fréquence, dans lequel on travaillait jusqu'à maintenant, vers le domaine de luminosité (l'image est en noir et blanc). Libu s'occupe de regrouper les blocs afin qu'ils forment des lignes pour envoyer au programme d'affichage. Le module doit aussi s'assurer de recommencer au haut de l'image quand assez de données ont été récoltées pour former une image, donc de passer à l'image suivante. RAMDAC est un programme simple qui s'occupe d'afficher l'image à l'écran. La compréhension du fonctionnement exact de l'application n'est pas nécessaire à la compréhension du reste du chapitre, mais ces explications sont données à titre informatif.

On peut voir un schéma des communications entre ces processus sur la figure 6.2.

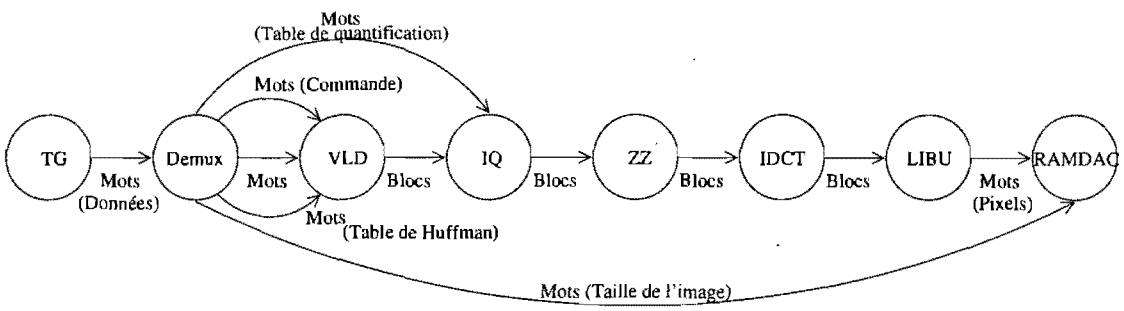


Figure 6.1: Schéma des communications de MJPEG

Le constructeur de ligne est modifié dans la version pour Mutek afin d'agir comme comparateur avec les valeurs précalculées sur la plateforme de référence et le RAMDAC sera enlevé. Certains changements sont aussi apportés au générateur de trafic afin

d'éviter l'emploi de systèmes de fichiers, qui sont loin d'être supportés.

Les résultats sont plutôt surprenants : quand on utilise seulement un PowerPC, GNU/-Linux prend seulement 11 secondes à exécuter l'application sur une entrée donnée, alors que Mutek prend plus d'une minute pour effectuer le même traitement. La cause probable de cette énorme différence est probablement due à une multitude de facteurs. Bien que Linux doit gérer plusieurs processus en plus de ceux de l'application devraient normalement réduire les performances. Cependant, la version de Mutek que j'utilise n'active pas l'unité de cache du processeur. Chaque lecture ou écriture se traduit donc comme un accès à la mémoire SDRAM externe, beaucoup plus lente que la mémoire cache située à l'intérieur du processeur. Cela a pour effet de réduire considérablement les performances. De plus, l'écriture à la console se fait en réservant un mutex et en faisant de l'attente active sur la console, ce qui a pour effet de miner encore davantage les performances attendues de Mutek. Les routines dépendantes du matériel que j'ai écrites sont peut-être aussi non optimales, mais elles affectent probablement moins le fonctionnement que les deux autres problèmes que je viens de citer.

Sous Mutek, l'application occupe 134 Ko en mémoire, système d'exploitation inclus. Sous GNU/Linux, la taille de nécessaire est près de 337 Ko avec une édition de liens statique ou 130 Ko avec une édition de lien dynamique, système d'exploitation non inclus. Dans tous les cas, 122 Ko sont occupés par les données de l'image à traiter. Le noyau de Linux occupe, avant démarrage, 934 Ko et l'image de disque, 3.3 Mo. GNU/Linux reporte 24 Mo de mémoire utilisé après le démarrage.

L'application roulant en mémoire seulement nous permet de faire une partie des scripts nécessaires pour permettre de faire rouler l'application sur différents systèmes d'exploitation. Le code de l'application roulant sous Mutek est identique à celui roulant sous GNU/Linux. Les seules différences sont les options du script de compilation. Cela n'a rien de surprenant et est surtout dû au fait que les deux systèmes d'exploitation supportent l'API POSIX. Les chaînes d'outils utilisées pour la compilation diffèrent.

6.3 Implémentation mixte

Avoir une application qui fonctionne sur les deux plateformes et dont la compilation ne diffère que par le script de compilation est intéressant, mais peu probant et loin du but que nous nous étions fixé. Il faut donc modifier cette application afin qu'elle soit un exemple de conception mixte en transformant un des modules en un matériel spécialisé.

Partant de la version logicielle, j'ai traduit le module du démultiplexeur en code VHDL. Ce module est loin d'être celui demandant le plus de temps de calcul, mais il est simple et facilement portable vers le matériel. Ce module étant celui qui fait le plus de communications, ce choix est probablement le pire possible. Cela nous offrira donc une borne supérieure sur la latence demandée par l'ajout de FIFO et les communications logicielles/matérielles. Donc, si l'on décide de passer un autre des modules communiquant avec le démultiplexeur en matériel, nous ne pouvons qu'alléger les communications. J'ai aussi développé un FIFO matériel pour faire le lien entre plusieurs modules matériels qui pourraient être développés.

6.3.1 Communications

À l'aide des outils de Xilinx, j'ai par la suite développé une interface pour le bus OPB afin de pouvoir faire la communication entre le matériel et le logiciel. Cette interface communique avec les modules matériels à l'aide des FIFO matériels que j'ai développés. Nous pouvons donc très facilement modifier la structure de la partie matérielle afin d'ajouter ou retirer un module. L'interface produit une série de registres projetés en mémoire qui correspond à chacun des FIFO entrants ou sortants des modules matériels. Si le FIFO a une largeur plus grande que celle du bus, l'interface concatène les mots reçus jusqu'à ce qu'on ait assez de données pour faire une écriture sur le FIFO. Des registres supplémentaires sont définis afin d'avoir un statut des FIFO (vides, pleins ou prêts) et un système de débogage du matériel. Le code de cette interface étant assez simple et répétitif, il serait très facile d'écrire un programme pour la générer automatiquement afin de refléter le nombre de FIFO de nos modules matériels.

Nous pourrions aussi étendre l'interface afin d'avoir une gestion des interruptions.

Cela augmenterait grandement l'efficacité de nos communications, mais demande une certaine réflexion pour savoir quand nous voulons générer une exception. Pour un FIFO qui écrit vers le matériel, nous pourrions vouloir générer une interruption lorsque le FIFO passe d'un état plein à un état non plein ou bien préférer en avoir une seulement lorsqu'il est vide. Le plus simple, si nous voulions supporter les exceptions, serait probablement de permettre de déclencher toutes les exceptions possibles, mais mettre un masque afin de permettre au programme de bloquer celle qu'il ne désire pas avoir. Cela permettrait au programme de demander d'être réveillé lorsqu'il attend une condition, au lieu de faire de l'attente active comme dans la version actuelle.

La figure 6.2 montre l'attente active, l'approche actuellement implantée. L'attente active est inefficace et consomme une grande quantité de temps processeur et occupe le bus inutilement, mais elle est la plus facile à implémenter. L'approche démontrée par 6.3, soit une gestion des interruptions au niveau du système d'exploitation n'est pas plus efficace puisqu'elle génère beaucoup d'interruptions qui nécessitent l'attention du système d'exploitation. Enfin, 6.4 nous présente l'approche avec un masque d'interruptions au niveau du matériel, qui est beaucoup plus efficace, mais beaucoup plus complexe à mettre en place.

L'annexe I fournit des pseudo-codes de l'exécution de ces différents modes.

6.4 Avantages et inconvénients

L'automatisation du flot étant loin d'être aussi complète que je l'aurais souhaité, les avantages de celui-ci sont plutôt restreints. Le principal est de pouvoir changer le système d'exploitation en modifiant seulement les arguments envoyés au compilateur. Le code remplacé pour la communication avec le matériel contient 3 appels à des fonctions pouvant être exécutées différemment selon le système d'exploitation : un appel pour l'initialisation du pilote de périphérique, un appel pour l'écriture à un FIFO et un appel pour les lectures à un FIFO. Les différentes versions de cet appel sont regroupées dans une bibliothèque pouvant être réutilisée pour d'autres applications et faisant partie intégrante du flot. Nous pouvons donc facilement compiler pour le système d'exploitation

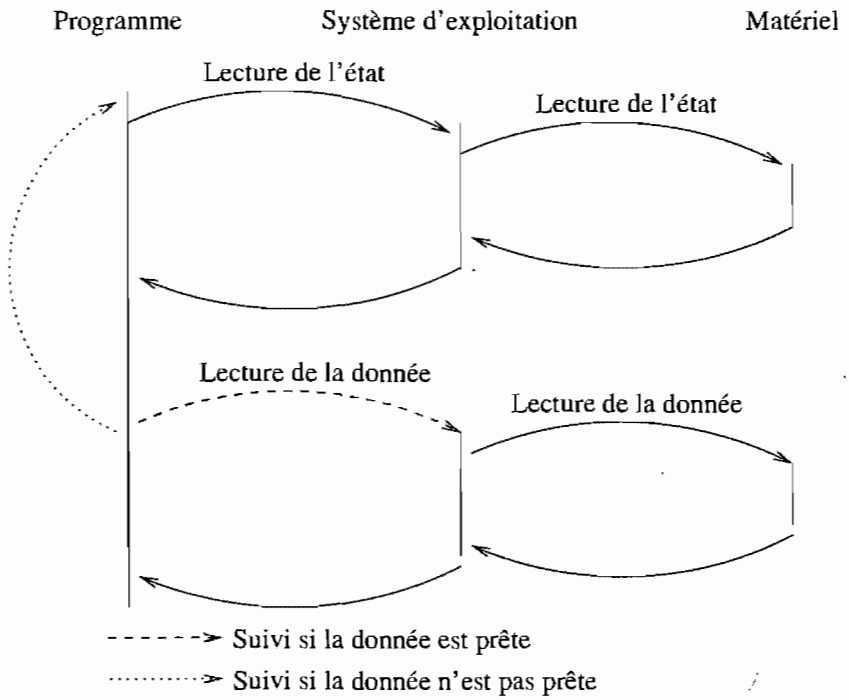


Figure 6.2: Gestion de l'attente, attente active

voulu afin de profiter des outils disponibles pour ce système d'exploitation.

Les choix d'implémentation que j'ai effectués ne supportent malheureusement pas les applications nécessitant du temps réel strict, ce qui peut être assez limitatif dans le domaine des systèmes embarqués, mais certaines applications utiles, comme MJPEG, peuvent être utilisées même sans temps réel strict.

En utilisant C, j'ai augmenté le niveau du langage habituellement utilisé par les concepteurs de matériel. On pourrait croire qu'on baisse aussi par la même occasion le niveau utilisé par les programmeurs, mais si on se fie à l'enquête du *Embedded Systems Design*, plus de 50 % des projets embarqués utilisent déjà C comme langage de programmation[10]. De plus, étant donné la facilité de changer de système d'exploitation, le débogage peut être effectué sur celui de notre choix, parmi ceux supportés par la plateforme et dont les scripts nécessaires ont été développés.

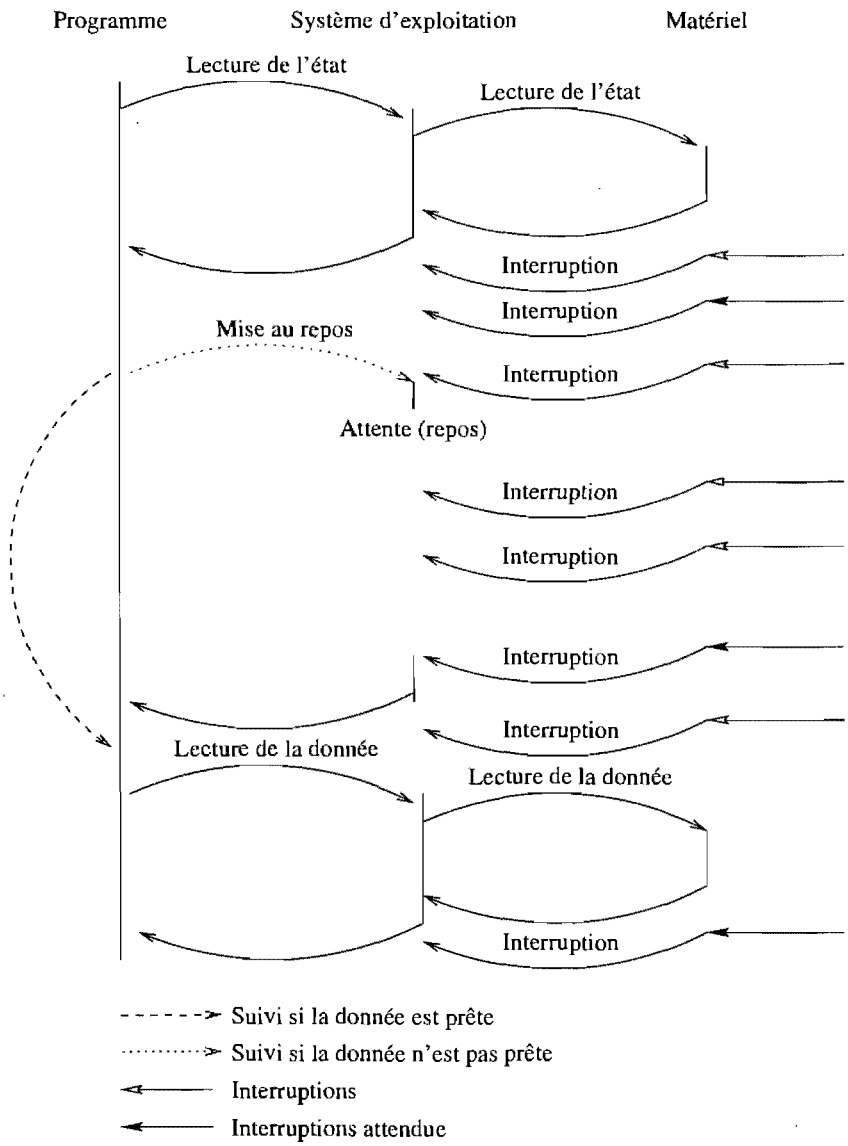


Figure 6.3: Gestion de l'attente, masque d'interruption logiciel

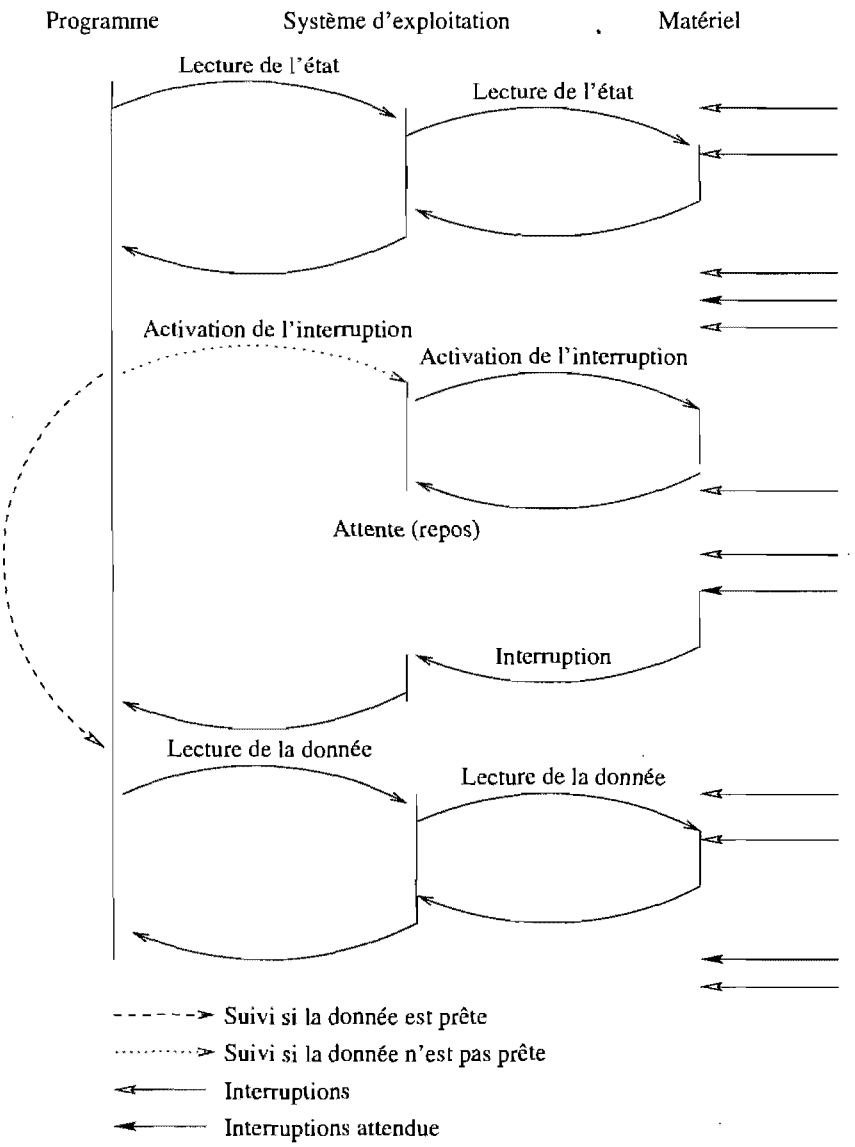


Figure 6.4: Gestion de l'attente, masque d'interruption matériel

CHAPITRE 7

FLOT DE DÉVELOPPEMENT

7.1 Motivation

Nous avons donc posé une cible à atteindre, sélectionné une plateforme qui puisse supporter nos expérimentations, adopté deux systèmes d'exploitation qui doivent être supportés, choisi un point de départ pour nous permettre d'atteindre notre but et une application pour valider nos raisonnements.

Il est donc temps de remplir les trous et proposer un flot qui nous permette d'aboutir à un système mixte logiciel/matériel tout en nous permettant de modifier le ou les systèmes d'exploitation utilisés à n'importe quelle étape de conception.

Le présent flot de développement contient malheureusement de nombreux trous et comporte plusieurs idées pour des travaux futurs.

7.2 Création

Comme nous avons vu précédemment, je suggère d'utiliser les processus communicants pour décrire le niveau comportemental. Pour générer un système complet, il faudrait alors écrire le comportement désiré en langage C en utilisant uniquement l'API supportée par le ou les systèmes d'exploitation présélectionnés et une librairie de FIFO. Dans le reste de ce chapitre, je vais tenir pour acquis que l'API en question est POSIX et la librairie, DPN [18].

7.3 Synthèse

Le présent mémoire ne couvre pas le processus de synthèse. Une extension très importante de mes travaux serait donc d'incorporer un outil de synthèse qui utilise C comme langage d'entrée. Comme nous avons vu au chapitre 5.2.4, certains outils permettent déjà cette synthèse. Cependant, ces outils ne sont peut-être pas adaptés à nos

besoins et devraient probablement être modifiés. Il faudrait aussi y intégrer une interface pour communiquer avec le FIFO matériel de notre module de communication. Certaines restrictions inhérentes à la synthèse pourraient aussi être ajoutées à notre langage source.

7.4 Partitionnement et communications

Puisque le but est de synthétiser le système complet automatiquement, le partitionnement peut facilement être modifié et retesté sans effort. Nous devons donc avoir un partitionnement initial, mais changer celui-ci est très facile et nous pouvons donc faire une exploration des différentes possibilités au moment voulu.

Pour l'instant, la façon de spécifier le partitionnement à la partie automatique du flot de développement est assez archaïque : il faut faire quelques modifications au code pour remplacer les parties matérielles et faire la communication vers le système d'exploitation.

Je prévois la création d'un fichier de configuration qui spécifie quels sont les processus matériels ainsi qu'une série de scripts qui permettraient d'automatiser la création des interfaces de communication. La création de ces scripts peut être assez compliquée. Comment déterminer la liste des FIFO accédés à partir d'un processus ? Nous pourrions suivre la liste des appels effectués aux méthodes *read* et *write* du FIFO, mais puisque ceux-ci peuvent être inclus dans des fonctions utilitaires, les appels peuvent être durs à retracer. Le plus simple est donc de trouver cette information dans le fichier de partitionnement. Le listing 7.1 est un exemple de ce que pourrait contenir ce fichier.

Listing 7.1: Example du format de fichier de partitionnement

```
//Nom          Type      NbFifoIn    NbFifoOut
threadTG       Logiciel  0           1
threadDemux    Materiel  1           5
threadVld      Logiciel  3           1
threadIq       Logiciel  2           1
threadZz       Logiciel  1           1
threadIdct     Logiciel  1           1
```

```
threadLibu     Logiciel 2          0
```

L'approche que je privilégie est d'inclure deux chiffres, soient le nombre de FIFO accédés en lecture et le nombre accédés en écriture, à la suite du nom du processus à synthétiser. Pour faciliter un peu plus le repérage et l'écriture des scripts, la structure passée en argument a la fonction du processus devrait commencer par les pointeurs vers les FIFO accédés en lecture, puis ceux accédés en écriture, et enfin les autres arguments, s'il y en a. Un exemple pour le module demux est donné dans le listing 7.2. Puisque le modèle de calcul choisi est les processus communicants, l'emploi d'arguments excédentaires est déconseillé et devrait plutôt être remplacé par un FIFO. Nous devons aussi détecter les FIFO qui sont uniquement matériels afin d'éviter de les générer en logiciel.

Listing 7.2: Structure de l'argument de threadDemux

```
typedef struct{
        FIFO *in[1];
        FIFO *out[5];
} demux_arg_t;
void *threadDemux(void * arg) {
        demux_arg_t fifos = (demux_arg_t *) arg;
        ...
```

Il faudrait aussi automatiser l'adaptation de notre module de communication, selon la largeur des FIFO et le nombre de celles-ci. Vu la simplicité de l'interface avec le bus OPB, l'automatisation ne devrait pas être très difficile. Un exemple de ce à quoi pourrait ressembler la fonction d'écriture du module est donné en annexe II. Cet exemple tient pour acquis que les FIFO sont de la même largeur que le bus et est énormément simplifié. ##NbFifoIn devrait aussi être remplacé afin de refléter le total du nombre de FIFO d'entrée présents dans le fichier de partitionnement.

Un travail futur important serait aussi de modifier l'interface de communication afin de supporter les interruptions. Le module matériel n'aurait qu'à ajouter des registres pour les masques d'interruptions et lancer une interruption à chaque fois qu'un changement d'état des FIFO survient et que le programme a demandé à être informé de ce change-

ment. Un registre mémorisant la dernière interruption lancée pourrait aussi être utile. Le pilote de périphériques devrait, quant à lui, être changé beaucoup plus profondément. Il faudrait permettre de faire dormir un processus qui est en attente d'un événement en attendant une interruption. Il faudrait aussi avertir le module matériel que nous voulons qu'il nous envoie une interruption lorsque la condition que nous attendons est satisfaite. Il faudrait aussi mettre en place une routine de traitement pour ces interruptions. Un pseudocode du traitement des interruptions est donné en annexe I.

7.5 Sélection du système d'exploitation

La sélection du système d'exploitation est une étape cruciale dans le développement d'un système logiciel/matériel. La méthodologie permet une plus grande latitude dans le choix du système d'exploitation et permet de le changer plus rapidement, si pour quelque raison que ce soit, le système d'exploitation initialement sélectionné doit être remplacé. En fait, un tel changement ne nécessiterait qu'une simple recompilation. Le script Makefile fourni à l'annexe III produit même deux versions pour les deux systèmes d'exploitation que nous avions décidé d'utiliser. Une modification dans le code peut donc être propagée tout de suite à chacun des systèmes d'exploitation. Un changement de partitionnement nécessite de modifier le code du module remplacé par un fil de d'exécution de threadCommHW pour chacun des FIFO utilisé par le module. Un pseudocode de cette modification est présenté dans l'annexe V Le code des fontions threadCommHW pour chacun des deux systèmes d'exploitation utilisé ainsi que la structure de l'argument à cette fonction est donnée à l'annexe IV.

Nous pouvons donc, si le temps et les ressources le permettent, explorer l'espace des systèmes d'exploitation disponibles et adaptés à notre application. Il faut aussi que le pilote de périphérique du FIFO matériel soit porté sur les systèmes d'exploitation sélectionnés et que les scripts d'automatisation sachent comment communiquer avec ce dernier et générer le code de remplacement adapté.

7.6 Compilation

Le processus de compilation risque d'être assez compliqué à automatiser puisque chaque système d'exploitation utilise sa propre chaîne d'outils adaptée. Quelques problèmes apparaissent aussi si on utilise les mêmes outils, dus à des architectures de système d'exploitation passablement différentes. Prenons, par exemple, le cas de nos deux systèmes d'exploitation : Mutek se compile en une bibliothèque avec laquelle notre application doit se lier et produit un exécutable binaire unique que nous pouvons charger sur la plateforme tandis que GNU/Linux est composé d'un ensemble de programmes que l'ont doit regrouper dans un paquetage. Nos programmes d'automatisation devraient être capables de prendre en compte ces particularités et un utilisateur qui désirerait rajouter un système d'exploitation à la liste de ceux supportés verrait donc sa charge de travail augmenter.

7.7 Chargement sur la plateforme

Le chargement sur la plateforme est, bien sûr, entièrement dépendant de la plateforme. Dans le cas qui nous occupe, plusieurs options s'offrent à nous, dont la plupart probablement disponibles sur la plupart des plateformes basées sur les FPGA de Xilinx. La méthode que j'ai le plus utilisée consiste à utiliser les fonctions du chargeur d'amorçage pour charger notre application par le port série, puis brancher vers les instructions d'initialisation de notre système d'exploitation. Cette façon nous assure qu'une initialisation du système a été faite, est simple et pourrait être facilement automatisée. Ces raisons font que c'est probablement la façon de faire la plus efficace pour la phase de test.

Une autre façon de faire serait d'inclure notre image dans la mémoire du FPGA, la BRAM. C'est l'option la plus efficace pour la redistribution puisque la configuration du FPGA inclut notre programme. Ainsi, l'utilisateur n'a rien à faire pour que l'application soit lancée. Le fichier de configuration du FPGA, ou *bitstream*, peut être chargé de différentes façons sur notre plateforme : par la mémoire flash sur la carte, par la mémoire CompactFlash externe et via JTAG. Les mémoires flash sont utiles pour la redistribution

puisqu'elles rechargent elles-mêmes le bitstream lors du redémarrage. La mémoire flash interne est plus compliquée à changer, ce qui peut être un avantage ou non selon les utilisations souhaitées. JTAG (ou IEEE 1149.1) est un protocole visant à faire des tests sur les circuits imprimés. Dans le cas de nos FPGA, le port JTAG nous permet aussi de faire le chargement de circuits sur le FPGA et pour le débogage des circuits que l'on y insère. Une primitive est présente dans la plateforme que nous utilisons pour communiquer avec le port de débogage des processeurs PowerPC. Le chargement via JTAG peut être utile pour tester notre combinaison logicielle/matérielle,

L'automatisation de ces méthodes de chargement serait relativement facile à réaliser, mais offre peu d'avantages vu la simplicité de la tâche.

7.8 Débogage

L'exécution à haut niveau ou la simulation de l'application avant de se lancer dans l'implantation devrait être en mesure de trouver la plupart des bogues, mais certains ne se révéleront qu'au moment de la mise en place. On peut penser, par exemple, aux communications avec le matériel déjà existant ou certaines concurrences non trouvées lors de la simulation. Il nous faut donc posséder des outils efficaces afin de faire le débogage de notre application une fois celle-ci en place. Les techniques utilisables varient selon le système d'exploitation et le niveau auquel on veut effectuer ce débogage.

7.8.1 Mutek

Un des problèmes majeurs rencontrés lors du développement au niveau système d'exploitation est le manque d'outils de débogage. Ces derniers utilisant souvent des ressources du système d'exploitation, ils deviennent donc difficiles à utiliser dans un environnement. Dans Mutek, l'application s'exécute au même niveau que le système d'exploitation, il peut donc être fastidieux de trouver les problèmes.

IBM fournit un simulateur pour le jeu d'instruction (*Intruction set simulator* ou ISS) pour le processeur PowerPC. Cependant, le reste du système lui est inconnu. La partie générée par le présent flot de développement pourrait assez facilement être modifiée pour

être compatible avec l'ISS : il faudrait seulement coder la partie de l'interface avec le bus. Cette façon de faire présente l'avantage de pouvoir être exécutée sur une machine plus performante que la plateforme cible, mais il est possible qu'elle ne détecte pas toutes les erreurs possibles.

Une autre solution consiste à implémenter un débogueur sur Mutek, ou du moins une souche (anglais : *stub*) afin de faciliter le débogage. Le peu de support pour les entrées/sorties de Mutek ne pose pas vraiment de problèmes puisqu'une souche pourrait communiquer avec la machine hôte par le port série déjà utilisable. Le débogueur pourrait aussi communiquer par réseau ou d'autres méthodes, mais en plus de coder la souche, il faudrait augmenter le support de matériel à Mutek. Un des grands avantages de cette méthode est la capacité de déboguer à distance, même une fois le système en place.

Il existe aussi une manière de brancher un débogueur directement sur le processeur PowerPC de la plateforme à l'aide d'outils fournis par Xilinx. Nous pouvons donc arrêter le fonctionnement d'un des deux processeurs pour inspecter le contenu de ses registres, faire un avancement pas-à-pas et autres manipulations fréquentes de débogage.

7.8.2 GNU/Linux

L'installation de GNU/Linux qui vient par défaut avec la plateforme n'offre pas d'outils de débogage. Cependant, il est relativement facile avec des scripts disponibles gratuitement sur Internet [16][23] de générer une nouvelle distribution qui contient les outils dont nous avons besoin. Nous pouvons donc ajouter un débogueur afin de faciliter la section utilisateur. Pour la section système, nous pouvons aussi utiliser la connexion JTAG pour contrôler directement le processeur et gérer le processus de débogage. Nous pouvons aussi utiliser les débogueurs disponibles pour le noyau, comme kdb ou kgdb.

Un développement sur une machine GNU/Linux nous permettrait aussi, contrairement à Mutek, d'avoir accès à un programme d'analyse (*tracer*) qui nous permette de détecter les modules qui prennent plus de temps à exécuter sur notre plateforme[9].

7.9 Communications avec l'extérieur

Le présent flot a quelques lacunes ; une d'entre elles étant la communication avec le monde extérieur. Une telle communication est réalisable si on étend un peu les restrictions lorsqu'on est sûr qu'un module va être gardé en logiciel. Cela nous amènerait, dans ce cas à permettre un accès plus grand aux commandes du système d'exploitation et aux matériels non générés sur le système. On pourrait aussi insérer un module factice que l'on indiquerait généré en matériel et le remplacer par un module généré d'une autre façon et qui communiquerait avec le monde extérieur. On pourrait simplifier l'inclusion d'un tel module en ajoutant une catégorie pour les modules factices dans nos programmes d'automatisation. Les scripts pourraient alors substituer le module par un code défini par l'utilisateur, idéalement dans un langage de description matériel et synthétisable. Nous pourrions aussi développer une librairie d'accès à des modules externes ou une interface pour accéder directement à des fils. Cette approche doit cependant inclure une synchronisation de bas niveau qui viendrait nuire à la simulation initiale.

L'accès au monde extérieur est donc une tâche ardue qui nécessiterait encore beaucoup de réflexion et qui enlèverait probablement certains avantages au présent flot. Cependant, ces communications extérieures sont cruciales pour à peu près n'importe quelle application réelle.

7.10 Temps réel

Une autre limitation importante du flot est qu'il ne supporte pas le temps réel. Il faudrait, pour le supporter, utiliser un API temps réel tel que μITRON, connaître les latences de nos interfaces de communication pour chacun des systèmes d'exploitation et, avant tout, s'assurer de l'exactitude de celles-ci. Cela offrirait un beau défi pour quelqu'un qui voudrait s'y attarder et serait grandement utile puisqu'une grande partie des applications embarquées nécessitent une borne sur le temps d'exécution de certaines fonctions, ce que le présent flot est incapable d'assurer.

7.11 Interface de programmation

Le présent flot nécessite malheureusement de fixer l'API tôt dans la conception. Ce choix limite malheureusement beaucoup le choix des systèmes d'exploitation que nous pouvons utiliser par la suite. Nous pourrions essayer d'abstraire l'API en utilisant des interfaces génériques pour les services offerts par le système d'exploitation. Une approche a déjà été étudiée dans [13]. L'approche est malheureusement basée sur le langage C#. Il faudrait donc probablement beaucoup de travail pour combiner les deux travaux.

7.12 Réalisé à ce jour

À ce jour, j'ai implémenté une partie du flot que je viens de présenter. Les outils fonctionnent sur un hôte GNU/Linux Fedora Core 7 avec processeur Xeon. Les outils devraient cependant fonctionner avec n'importe quelle plateforme supportant les outils de développement GNU. Le système cible est la carte Amirix AP-1100 avec une configuration incluant un PowerPC sur un bus PLB, la primitive de débogage JTAG, 64 Mo de mémoire SDRAM DDR, un contrôleur série RS-232. Cette configuration est illustrée sur la figure 3.1 et la disposition de la mémoire sur la figure 3.2.

La synthèse doit être faite à la main à partir du fichier C. Bien que l'interface branchant les modules matériels au bus OPB soit réalisée et relativement simple, l'adaptation de celle-ci au nombre et à la direction des FIFO communiquant avec le module matériel doit aussi être faite manuellement. Une fois le module matériel créé, celui-ci doit être intégré au reste de la plateforme à l'aide de l'outil *Xilinx Platform Studio (XPS)*. Le tout est chargé sur le FPGA à l'aide d'un câble JTAG et le logiciel *Impact* de Xilinx.

Du côté du logiciel, les deux systèmes d'exploitation supportés sont la version de Mutek que j'ai modifiée pour être exécutable sur les processeurs PowerPC et un GNU/-Linux avec un noyau version 2.4.18, modifié par Amirix pour s'exécuter sur leur carte, et une distribution ELDK 3.0. J'ai un fichier Makefile qui compile chacun vers un système d'exploitation. Ce fichier est donné en annexe III. La compilation des deux versions diffère essentiellement sur l'emplacement des librairies et par les lignes de compilation. Ces dernières spécient de ne pas utiliser la librairie standard, mais bien les librairies de

Mutek. Le module matériel que j'ai implémenté contient encore quelques bogues qui rendent difficiles les tests d'efficacité.

Le langage utilisé est C et l'API testé est un POSIX de base avec des primitives pour les fils d'exécution et de synchronisation *PThread*. Les outils disponibles actuellement pourraient sûrement supporter un plus grand ensemble, mais Mutek offre un sous-ensemble assez restreint de POSIX. Nous ne pouvons alors pas vraiment supporter plus de fonctions, à moins de changer de système d'exploitation. L'API est étendu avec le support de FIFO offert par la bibliothèque DPN, qui vient du projet Disydent[18]. La bibliothèque prévoit des primitives pour établir des FIFO entre le logiciel et le matériel, mais puisque nous voulons que la programmation soit le plus indépendante du matériel possible, ces primitives ne devraient pas être utilisées lors de la conception initiale du système. Nous remplacerons plutôt le fil d'exécution qui s'occupait d'exécuter la fonction que nous avons traduite en matériel par plusieurs fils d'exécution qui s'occuperont de transférer les communications vers le matériel. Un pseudocode et un script Bash pour effectuer cette tâche sont présentés à l'annexe V. Une partie du code qui s'occupe de transférer les données vers le système d'exploitation ou directement vers le matériel, dans le cas de Mutek, est donné à l'annexe IV.

Des pilotes pour le matériel ont aussi été développés afin que les deux systèmes d'exploitation puissent communiquer avec le matériel. Mutek étant très simpliste, celui-ci consiste seulement en une boucle faisant le lien entre un FIFO logiciel et celui correspondant en matériel. Une partie du code du pilote est fourni dans l'annexe IV pour Mutek et dans l'annexe VI pour celui de Linux. L'intérieur de la boucle consiste à vérifier le statut du FIFO qui lui est associé. Si celui-ci est prêt, on échange quatre octets de données avec le périphérique. Si le périphérique n'est pas prêt, nous passons le contrôle du processeur à un autre fil d'exécution. Un fil d'exécution incluant un appel à la fonction contenant cette boucle doit être effectué pour chaque FIFO communiquant des données entre le matériel et le logiciel. Cette liaison doit être effectuée manuellement en passant une structure de donnée contenant l'adresse de base du module matériel, le décalage du registre de statut, le décalage du registre du FIFO, un masque indiquant quel bit de statut observer, le FIFO logiciel, le sens du FIFO (lecture ou écriture), la largeur du FIFO.

Normalement chaque FIFO ne devrait pas être utilisé par plus d'un fil d'exécution, ce qui rend inutile l'emploi de mutex. Cependant, celui-ci ne rajoute pas beaucoup de délais d'exécution et offre plus de latitude aux utilisateurs.

Un pilote de matériel a aussi été développé pour Linux. Ce module est un pilote de classe caractère basé sur le pilote exemple scull. Ce dernier est défini dans le livre *Linux Device Drivers*, 2e édition[24]. Le pilote développé a été compilé pour être chargé avec la version du noyau Linux présent sur notre plateforme. Le pilote crée une série de fichiers de périphériques. Le nombre de fichiers de périphériques ainsi créés dépend d'un fichier de configuration et d'un argument envoyé au noyau lors du chargement du module. Sous Linux, ces fichiers de périphériques sont les moyens privilégiés pour les communications entre les logiciels utilisateurs et systèmes. Ils contiennent un numéro de périphérique majeur, qui détermine avec quel pilote communiquer, et un mineur, qui spécifie avec quelle entité gérée par le pilote nous voulons communiquer. Dans le cas qui nous intéresse, le majeur peut être décidé par le noyau Linux parmi ceux encore disponibles, stipulé à la compilation ou précisé au chargement. Le mineur contient l'index, en entier, du FIFO par rapport à l'adresse de base de notre module matériel et, sur le bit supérieur, la direction du FIFO.

Lors d'une tentative d'écriture sur le périphérique, le pilote vérifie la disponibilité du périphérique et, s'il est disponible, écrit quatre octets (un entier 32 bits) sur le périphérique. Si le périphérique n'est pas prêt, le module se contente de retourner le contrôle à l'application en spécifiant qu'il n'a rien écrit sur le périphérique. Ces accès au matériel sont protégés par un verrou à attente active pour s'assurer de la cohérence des accès aux périphériques. Le côté utilisateur, quant à lui, lit une entrée sur le FIFO logiciel et fait une écriture dans le fichier représentant le périphérique. Les lectures suivent les mêmes principes. Si la tentative a échoué, il laisse le processeur à un autre fil d'exécution et réessaie plus tard.

En raison du fait que le module matériel est encore défectueux, je n'ai pas pu tester les pilotes en profondeur. Cependant, le pilote de périphérique de Mutek étant très simple et se limitant à effectuer des lectures et écritures vers le périphérique, le fonctionnement est facile à valider. Celui pour Linux est plus difficile à valider du fait qu'il

fait quelques déplacements entre la mémoire du programme, la mémoire du noyau et la mémoire physique. L'initialisation et les fonctions nécessaires aux pilotes sous Linux, comme l'ouverture du fichier de périphérique, rendent aussi les erreurs plus difficiles à débusquer.

Le chargement sur la plateforme se fait manuellement. Sous Mutek, j'effectue le chargement du système d'exploitation en même temps que celui du programme par le port série à l'aide du chargeur de démarrage U-Boot. Le fichier généré par le Makefile est du format SREC et contient donc une série de lignes incluant de l'information à charger et l'adresse à laquelle effectuer l'opération. Une fois le programme en mémoire, nous demandons au chargeur de démarrage de brancher à la première instruction de l'initialisation du système d'exploitation. Cette adresse est spécifiée dans le script d'édition de liens. Une fois Mutek initialisé, il branche directement au programme.

Sous GNU/Linux, le chargement s'effectue en mettant le programme et le pilote de périphérique sur une carte CompactFlash. Une fois cette mémoire insérée sur la carte Amirix, l'accès peut s'effectuer à partir du GNU/Linux de la plateforme. Ce GNU/Linux est démarré par défaut si aucune intervention n'a eu lieu lors du démarrage. Le module est chargé à l'aide de la commande insmod et il faut appeler le programme lorsque nous voulons l'exécuter.

CHAPITRE 8

CONCLUSION

Le domaine du codesign est encore en pleine effervescence. Beaucoup d'améliorations restent encore possibles et devront être faites si l'on veut pouvoir continuer d'accélérer le rythme de production des systèmes électroniques.

Ma recherche apporte sa contribution en ajoutant plus de flexibilité quant à la sélection du système d'exploitation au flot de conception normalement adopté pour le codesign. Pour y arriver, nous avons commencé par fixer la plateforme. Nous avons donc choisi une carte Amirix AP-1100 avec FPGA Virtex-II Pro qui implémente un circuit incluant un processeur PowerPC. Ensuite, nous avons aussi fait un survol de plusieurs systèmes d'exploitation offerts sur le marché et arrêté un choix sur deux passablement différents : Mutek et GNU/Linux. Nous avons aussi regardé le potentiel de quelques langages de programmation afin de fournir un langage de départ pour le codesign. Suite à cette analyse, nous avons décidé de continuer avec le langage C.

Nous avons donc basé notre flot sur les processus communicants et le langage C afin de concevoir un système mixte logiciel et matériel à l'aide d'un seul langage. Une fois le système créé, nous pouvons raffiner les modules qui finiront en matériel et laisser le système créer la partie logicielle et les communications. Le flot permet de changer le système d'exploitation utilisé à l'aide d'une seule commande *make*, pourvu que celui-ci supporte l'API sélectionné ainsi que le langage C. Certaines tâches sont aussi nécessaires afin qu'un nouveau système d'exploitation soit utilisable avec mes outils.

J'ai donc implémenté une partie de ce flot pour supporter Mutek et GNU/Linux sur la carte Amirix. La traduction du modèle vers un langage de description matériel se fait manuellement. Le flot supporte une partie de l'API POSIX et utilise la bibliothèque DPN pour fournir la communication interprocessus. Le développement de pilote de périphérique a été nécessaire afin de supporter les deux systèmes d'exploitation. Le chargement sur la plateforme se fait aussi manuellement. L'implémentation ne supporte pas les primitives de temps réel et la communication avec d'autres modules matériels, possible-

ment obtenus par d'autres méthodologies ou provenant de tierces parties, doit se faire à la main.

Afin de tester mon flot et aider à tester les outils, j'ai utilisé l'application MJPEG. Une version logicielle m'a été fournie et je l'ai adaptée afin d'enlever la nécessité d'avoir un système de fichiers. J'ai aussi traduit manuellement une partie dans un langage de description matériel afin de développer les communications avec le matériel.

En conclusion, de nombreuses avancées sont encore possibles dans le domaine de la conception électronique assistée par ordinateur, mais j'y aurais apporté ma modeste participation en permettant une plus grande flexibilité quant à l'intégration du système d'exploitation dans le flot de développement logiciel/matériel.

BIBLIOGRAPHIE

[1] Amirix System Inc. AP1000 FPGA development board users guide, 2005. `http://www.amirix.com`.

[2] R.M. Barned et R.J. Richards. Uniform Driver Interface (UDI) reference implementation and determinism. *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE*, pages 301–310, 2002. ISSN 1080-1812.

[3] Matthias Bauer et Wolfgang Ecker. Hardware/software co-simulation in a VHDL-based test bench approach. Dans *DAC '97 : Proceedings of the 34th annual conference on Design automation*, pages 774–779, New York, NY, USA, 1997. ACM. ISBN 0-89791-920-3.

[4] E. Bergeron, M. Feeley et J.P. David. Toward on-chip JIT synthesis on Xilinx VirtexII-Pro FPGAs. *Circuits and Systems, 2007. NEWCAS 2007. IEEE Northeast Workshop on*, pages 642–645, Aug. 2007.

[5] E. Bergeron, X. Saint-Mleux, M. Feeley et J.P. David. High level synthesis for data-driven applications. *Rapid System Prototyping, 2005. (RSP 2005). The 16th IEEE International Workshop on*, pages 54–60, June 2005. ISSN 1074-6005.

[6] Étienne Bergeron. *Compilation efficace pour FPGA reconfigurable dynamiquement.* Thèse de doctorat, Université de Montréal, 2008.

[7] J. Chevalier, M. de Nanclas, L. Filion, O. Benny, M. Rondonneau, G. Bois et El Mostapha Aboulhamid. A SystemC refinement methodology for embedded software. *Design & Test of Computers, IEEE*, 23(2):148–158, March-April 2006. ISSN 0740-7475.

[8] Pai H. Chou, Ross B. Ortega et Gaetano Borriello. The chinook hardware/software co-synthesis system. Dans *ISSS '95 : Proceedings of the 8th international sympo-*

sium on System synthesis, pages 22–27, New York, NY, USA, 1995. ACM. ISBN 0-89791-771-5.

[9] M. Desnoyers et M. Dagenais. The LTTng tracer : A low impact performance and behavior monitor for GNU/Linux. Dans *OLS (Ottawa Linux Symposium) 2006*, pages 209–224, 2006.

[10] EEtimes. 2006 state of embedde market survey. *Embedded Systems Design*, 2006. `ftp://ftp.embedded.com/pub/ESD%20SubscribSurvey/2006%20ESD%20Market%20Study.pdf`.

[11] Esys.net. ESys.Net. `http://www.esys-net.org/`.

[12] International Technology Roadmap for Semiconductors. 2008 update overview. 2008. `http://www.itrs.net/Links/2008ITRS/Update/2008_Update.pdf`.

[13] Bruno Girodias. Une plateforme pour le raffinement des services d'OS pour les systèmes embarqués, 2005.

[14] Richard Goering. EDA '07 forecast: string, but watch the bumps. *EE Times*, 2007. `http://www.eetimes.com/news/latest/showArticle.jhtml?articleID=196800350`.

[15] T. Katayama, K. Saisho et A. Fukuda. Prototype of the device driver generation system for UNIX-like operating systems. *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, pages 302–310, 2000.

[16] Dan Kegel. Building and testing gcc/glibc cross toolchains. `http://www.kegel.com/crosstool`.

[17] S. Lemon et K. Rossi. An object oriented device driver model. *Compcon '95.'Technologies for the Information Superhighway', Digest of Papers*, pages 360–366, Mar 1995. ISSN 1063-6390.

[18] LIP6-ASIM. **Di**gital **s**ystem **d**esign **en**vironment. http://www-asim.lip6.fr/recherche/disydent/.

[19] Gordon E. Moore. Cramming more components onto integrated circuits. 1965. http://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf.

[20] M. O'Nils, J. Oberg et A. Jantsch. Grammar based modelling and synthesis of device drivers and bus interfaces. *Euromicro Conference, 1998. Proceedings. 24th*, 1:55–58 vol.1, Aug 1998.

[21] Open SystemC Initiative (OSCI). SystemC user guide, 2001. http://www.systemc.org.

[22] Ian Page. Hardware-software co-synthesis research at Oxford. 1996.

[23] pengutronix. PTXdist - Reproducable Embedded Linux Systems. http://www.pengutronix.de/software/ptxdist/index_en.html.

[24] Alessande Rubini et Jonathan Corbet. *Linux Device Drivers, 2nd Edition.* O'REILLY, 2001.

[25] Richard Stallman. UDI and free software, 1998. http://www.linuxtoday.com/developer/19981005002050P.

[26] TIMA-SLS. **AP**plication **e**lements for **s**oc. http://code.google.com/p/apes-elements/.

[27] Jim Turley. Operating systems on the rise. *Embedded Systems Design*, 2006. http://www.embedded.com/columns/surveys/187203732.

Annexe I

Pseudo-code gestion de l'attente

Listing I.1: Gestion de l'attente attente active

```
Pilote:
  Écriture():
    Tant que écriture non prête
      matériel_vers_pilote( Status )
    fin Tant que

    fifo_utilisateur_vers_pilote( données )
    pilote_vers_matériel( données )
```

Listing I.2: Gestion de l'attente masque d'interruption logiciel

```
Matériel:
  Traiter_status():
    Pour chaque bits activés faire
      lancer interruption

Pilote:
  Interruption():
    Si interruption non masquée
      Si interruption de lecture
        matériel_vers_pilote( données )
        pilote_vers_fifo_utilisateur( données )
      Sinon interruption d'écriture
        fifo_utilisateur_vers_pilote( données )
        pilote_vers_matériel( données )
      fin Si
    fin Si

    Réinitialiser Masque
    Effacer interruption
    Notifier logiciel utilisateur

  Écriture():
    matériel_vers_pilote( Status )
    Si écriture prête
      fifo_utilisateur_vers_pilote( données )
      pilote_vers_matériel( données )
    Sinon
      Repos en attente d'interruption
      Exécuter un autre module
```

Listing I.3: Gestion de l'attente masque d'interruption matériel

```
Matériel:
  Traiter_status():
    Pour chaque bits activés faire
      Si interruption non_masqué
        lancer interruption
      fin Si
    fin Pour
  Changer_Masque(nouveau_masque):
    masque = nouveau_masque

Pilote:
  Changer_Masque(nouveau_masque):
    logiciel_vers_matériel(nouveau_masque)

  Interruption():
    Si interruption de lecture
      matériel_vers_pilote( données )
      pilote_vers_fifo_utilisateur( données )
    Sinon interruption d'écriture
      fifo_utilisateur_vers_pilote( données )
      pilote_vers_matériel( données )
    fin Si

    pilote_vers_matériel( nouveau_masque )
    Effacer interruption
    Notifier logiciel utilisateur

  Écriture():
    matériel_vers_pilote( Status )
    Si écriture prête
      fifo_utilisateur_vers_pilote( données )
```

```
    pilote_vers_matériel( données )
Sinon
    pilote_vers_matériel( nouveau masque )
    Repos en attente d'interruption
    Exécuter un autre module
```

Annexe II

Partie du module d'interface matériel

```
SLAVE_REG_WRITE_PROC : process ( Bus2IP_Clk ) is
  variable ack;
  variable old_sel;
  if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
    if Bus2IP_Reset = '1' then
      ack := false;
    else
      GEN_WRITE_IF: for I in 1 to ##NbFifoIn generate
        --slv_reg_write_sel is one hot selector
        if slv_reg_write_sel(I) = '1' and ack = false then
          old_sel(I) := '1' ;
          if fifo_full(I) then
            err := true;
          else
            ack := true;
            fifo_data(I) <= Bus2IP_data;
            fifo_wr(I) <= '1';
          end if;
        elsif old_sel(I) = '1' and
            not slv_reg_write_sel (I) = '1' then
            --le maitre a recu notre message
          if fifo_wr_ack(I) = '1' then
            old_sel(I) := '0';
            ack := false;
            fifo_wr(I) <= '0';
          elsif err = true then
            err := false
```

```
            end if;
          end if;
        end GEN_WRITE_IF;
      end if;
      write_ack <= ack;
      write_err <= err;
    end if;
end process SLAVE_REG_WRITE_PROC;
```

Annexe III

Makefile

```
DDIR_MUTEK = ./mutek_ddir
DDIR_LINUX = ./linux_ddir
CC = ppc_4xx-gcc
CFLAGS_MUTEK  = -nostdlib -g -G0 -O0 -finline-limit=10000 \
    -I$(DDIR_MUTEK)/include -I. -DCASS -Wall
LDFLAGS_MUTEK = -L$(DDIR_MUTEK)/lib -lpthread -Tpowerpc.sc \
    -lc -ldpn -lpthread -lhandler
CFLAGS_LINUX  = -g -G0 -O0 -finline-limit=10000 \
    -I$(DDIR_LINUX)/include -I. -DCASS -Wall
LDFLAGS_LINUX = -L$(DDIR_LINUX)/lib -lpthread  -lc -ldpn \
    -lpthread
OBJ     = tg \
          demux  \
          vld  \
          libu  \
          huffman iq parse zz idct \
          table_vld utils HighLevelChannel soft-p
ELF_OBJ = output.elf img.elf
test:
  echo allo $(addsuffix .mutek.o,$(OBJ))

mutek: $(addsuffix .mutek.o,${OBJ}) ${ELF_OBJ}
  ${CC} -o $@ $(addsuffix .mutek.o,${OBJ}) ${ELF_OBJ} \
      ${LDFLAGS_MUTEK}

linux: $(addsuffix .linux.o,$(OBJ)) ${ELF_OBJ}
  ${CC} -o $@ $(addsuffix .linux.o,$(OBJ)) ${ELF_OBJ} \
```

```
      ${LDFLAGS_LINUX}

all : mutek linux

img.elf:
  ppc_4xx-objcopy --rename-section .data=.img -I binary \
      -O elf32-powerpc -B powerpc:common img.jpg img.elf

output.elf: output.bin
  ppc_4xx-objcopy --rename-section .data=.img -I binary \
    -O elf32-powerpc -B powerpc:common output.bin output.elf

output.bin:
  $(MAKE) -C reference
  reference/jpeg-p img.jpg

%.mutek.o : %.c
  $(CC) $(CFLAGS_MUTEK) -c $< -o $@

%.linux.o : %.c
  $(CC) $(CFLAGS_LINUX) -c $< -o $@

realclean: clean
  rm -f jpeg
clean :
  rm -f $(addsuffix .mutek.o,$(OBJ)) \
      $(addsuffix .linux.o,$(OBJ)) output.bin linux mutek

.PHONY: clean test
```

Annexe IV

Partie du module de communication logiciel

Listing IV.1: structure de l'argument à threadCommHW

```
struct commHW{
  void *base_addr;
  int offset_device;
  int offset_status;
  Channel *chan;
  int type;
  int mask;
  size_t size;
};
```

Listing IV.2: threadCommHW pour mutek

```
void threadCommHW(void *args){
  struct commHW *c=(struct commHW *) args;
  int data;
  volatile int* device= (int *) base_addr+offset_device;
  volatile int* status= (int *) base_addr+offset_status;
  Channel * chan = c->chan;

  /* Les canaux haut niveau sont utilisé lorsque
   * la FIFO est plus large que le bus (HL) */
  HLChannel hlc;
  HLchannelInit(c->chan,&hlc);

  if(c->type == WRITE_HL){
    while(1){
      if(!(c->mask & *(c->status))){
        HLchannelRead(&hlc,&data,c->size);
        (*c->device) = data;
      }else{
        sched_yield();
      }
    }
  }else if(c->type == READ_HL){
    while(1){
      if(!(c->mask & *(c->status))){
        data = *c->device;
        HLchannelWrite(&hlc,&data,c->size);
      }else{
        sched_yield();
      }
    }
  }else if(c->type == WRITE){
```

```
    while(1){
      if(!(c->mask & *(c->status))){
          data =0;
          channelRead(chan,&data,1);
          (*c->device) = data;
      }else{
        sched_yield();
      }
    }
  }else if(c->type == READ){
    while(1){
      if(!(c->mask & *(c->status))){
        data = *c->device;
        channelWrite(c->chan,&data,1);
      }else{
        sched_yield();
      }
    }
  }
}
```

Listing IV.3: threadCommHW pour GNU/Linux

```
void threadCommHW(void *args){
  struct commHW *c=(struct commHW *) args;
  char *path[50];
  snprintf(path,50,"/dev/fifo%i",c->offset_device);
  void *data[c->size];
  size_t nbytes;
  size_t result;
  Channel *chan = c->chan;
  /* Le pilote se charge de diviser en morceaux
   * si les données sont plus larges que le bus */
  if(c->type == WRITE_HL || c->type == WRITE){
    int fd = open(path,O_WRONLY);
    if (fd <= 0){
      return -1;
    }
    while(1){
      channelRead(chan,data,1);
      nbytes = 0;
      while(nbytes < c->size){
        result=write(fd,data+nbytes,c->size-nbytes);
        if (result >0){
          nbytes += result;
        }else{ //L'écriture a échoué
          sched_yield();
        }
      }
    }
  }else if(c->type == READ_HL || c->type == READ){
    int fd = open(path,O_RDONLY);
    if (fd <= 0){
      return -1;
```

```
    }
    while(1){
      nbytes = 0;
      while(nbytes < c->size){
        result=read(fd,data+nbytes,c->size-nbytes);
        if (result >0){
          nbytes += result;
        }else{ //La lecture a échoué
          sched_yield();
        }
      }
      channelWrite(chan,data,1);
    }
  }
}
```

Annexe V

Automatisation de l'intégration des communications

Un fichier de configuration contient la liste des processus à traduire en matériel ainsi que le nombre de FIFO qui communiquent avec lui en entrée et en sortie. Un exemple d'un tel fichier est donné dans le listing 7.1. Le pseudocode pour enlever les FIFO qui sont seulement matériels n'est pas donné.

Listing V.1: Pseudo code pour l'automatisation des communications

```
Pour chaque module dans fichier_partition faire
  Si Type de module est Matériel faire
    Enlever Files seulement en matériel
    rechercher ligne creation du fil d'exécution
    remplacer la ligne par
     1. création de NbFifoIn structure commHW de type READ_HL
     2. création de NbFifoOut structure commHW de type WRITE_HL
     3. appel de NbFifoIn + NbFifoOut création de fil
          d'exécution exécutant threadCommHW avec les stucture
          créées en 1. et 2.
    rechercher la lignes d'allocation de
      structure de fil d'exécution
    remplace la ligne par NbFifoIn + NbFifoOut structure
       de fil d'exécution
```

L'implémentation Bash est naïve et ne traite pas les cas ou les files sont uniquement en matériel, ni les cas ou la largeur des FIFO est différente de 32 bits.

Listing V.2: Script Bash implementant l'automatisation

```
#!/bin/bash
file=partition
for a in $(grep -E '^[a-zA-Z]+' $file | cut -f 1 -d' ');
do
  read nom type NbFifoIn NbFifoOut < <(grep $a $file |
    sed 's/[ ]+/ /g' )
  if [ "t$type" == "tMateriel" ]; then
    echo "({\
  struct commHW *In=malloc(sizeof(struct commHW)*$NbFifoIn);\
  int i =0;\
  Channel **chan=##new_chan;\
  for(i=0;i<$NbFifoIn;i++){\
   In[i].base_addr=0x4800000;\
   In[i].offset_device=i+1;\
   In[i].offset_status=0;\
   In[i].chan=chan[i];\
   In[i].type=READ_HL;\
   In[i].mask=1<<(2*i);\
   In[i].size=4;\
   pthread_create(\&${nom}__pthread_t[i],NULL,\
     \&threadCommHW,\&In[i]);\
  }\
  struct commHW *Out=malloc(sizeof(struct commHW)*$NbFifoOut);\
  for(i=0;i<$NbFifoOut;i++){\
   Out[i].base_addr=0x4800000;\
   Out[i].offset_device=i+1;\
   Out[i].offset_status=0;\
   Out[i].chan=chan[i+$NbFifoIn];\
```

```
   Out[i].type=WRITE_HL;\
   Out[i].mask=1<<(2*i+1);\
   Out[i].size=4;\
   pthread_create(\&${nom}__pthread_t[i],NULL,\
    \&threadCommHW,\&Out[i]);\
   }\
${nom}__pthread_t[0];})" > .tmppartition

  for b in $(grep -l -E \
      "pthread_create([^,]*,[^,]*,.*$nom.*)" *.c); do
    read ptht newchan < <(grep -E \
        "pthread_create([^,]*,[^,]*,.*$nom.*)" $b |
     sed "s/^.*pthread_create([^,A-Za-z]*\([A-Za-z]*\)[^,A-Za-z\
        ]*,[^,]*,[^,]*$nom[^,]*,\([^)]*\)).*$/\1 \2/")
    sed -i "s/##new_chan/$newchan/" .tmppartition;
    sed -i "s/pthread_create([^,]*,[^,]*,[^,]*$nom[^)]*)/\
      $(cat .tmppartition)/" $b
    sed -i "s/\(pthread_t .*$ptht.*;\)/\1
      ${nom}__pthread_t[$NbFifoIn+$NbFifoOut]/" $b
    rm -f .tmppartition
  done
  fi
done
```

Annexe VI

Pilote de périphérique

Listing VI.1: Partie du pilote de périphérique sous linux

```
ssize_t scull_read(struct file *filp, char *buf, size_t count,
                   loff_t *f_pos)
{
    Scull_Dev *dev = filp->private_data;
    ssize_t ret = 0;
    int status,data;
    count=1;

    if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
    rmb();
    status = readl(dev->status);
    if(!(status & dev->rmask)){
      rmb();
      data = readl(dev->device);
      if(copy_to_user(buf, &data, count)){
        ret=-EFAULT;
        goto out;
      }
    }

    ret = count;

 out:
    up(&dev->sem);
    return ret;
```

```
}
ssize_t scull_write(struct file *filp, const char *buf,
                size_t count, loff_t *f_pos)
{
    Scull_Dev *dev = filp->private_data;

    ssize_t ret = 0;
    int status,data;
    count=1;

    if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
    status = readl(dev->status);
    rmb();
    if(!(status & dev->rmask)){
      if(copy_from_user( &data, buf, count)){
        ret=-EFAULT;
        goto out;
      }
      writel(data,dev->device);
    }

    ret = count;

 out:
   up(&dev->sem);
   return ret;
}
```