MITSUBISHI ELECTRIC
INFORMATION TECHNOLOGY EUROPE - TCL
Mitsubishi Electric ITE-TCL
1, allée de Beaulieu
CS 10806 – 35708 Rennes CEDEX 7 – France
Tel. +33 (0) 2 23 45 58 58

ST50 – Projet de fin d'études

Noyau Linux et multi-processeur pour une plateforme embarquée

Thomas Petazzoni – GI06

Semestre d'automne 2004-2005

Suiveur en entreprise: M. David Mentré
Suiveur UTBM: M. Jean-Charles Créput

Remerciements

Je tiens à remercier la société *Mitsubishi Electric ITE-TCL* de m'avoir permis d'effectuer ce stage au sein de son équipe *Protocoles*.

Je souhaite en particulier remercier Madame Isabelle Moreau, responsable de l'équipe, et Monsieur David Mentré, mon suiveur, pour avoir supervisé mon travail. Je remercie également l'ensemble de l'équipe pour le temps qu'ils m'ont consacré durant le stage.

Je remercie également Monsieur Mathieu Bertrand pour ses explications sur le fonctionnement des logiciels et du matériel, Monsieur Laurent Foube pour les nombreux échanges concernant les technologies matérielles utilisées, ainsi que Monsieur Christophe Mangin qui dirige les développements du projet *FlexNP*.

Enfin, je remercie l'ensemble du personnel de la société *Mitsubishi Electric ITE-TCL* pour son accueil.

Table des matières

Introduction

J'ai effectué mon projet de fin d'études *ST50* au sein de l'entreprise *Mitsubishi Electric ITE-TCL* du 6 septembre 2004 au 18 février 2005. Le sujet initial était la génération de *ramdisk*, le portage du noyau Linux 2.6 et la mise en place du fonctionnement multi-processeur pour une plateforme embarquée.

Après une courte présentation de l'entreprise (partie 1), le contexte technique du stage sera détaillé (partie 2). Les parties suivantes détailleront les grandes étapes du travail réalisé pendant le stage : génération de chaînes de cross-compilation et de *ramdisk* (partie 3), portage du noyau Linux 2.6 (partie 4) et mise en place du fonctionnement multi-processeur (partie 5).

Ce rapport constitue à la fois un compte-rendu du travail effectué durant mon stage et une documentation technique à destination de l'équipe. Il contient donc des informations techniques précises concernant le travail réalisé.

Chapitre 1

Présentation de l'entreprise

1.1 Mitsubishi Electric

Mitsubishi Electric est un grand groupe international qui comptait en 2003 plus de 116 000 salariés et réalisait un chiffre d'affaires de 28 milliards d'Euros. L'entreprise, fondée en 1921, est l'un des plus importants groupes industriels au monde. Il est issu du conglomérat industriel et financier japonais *Mitsubishi* dont les activités furent séparées après la seconde guerre mondiale pour constituer des pôles d'activités distincts. *Mitsubishi Electric* est un de ces 29 pôles qui couvrent des domaines aussi divers que la banque, l'assurance, l'automobile, le pétrôle, la chimie, le nucléaire, le transport, l'immobilier ou encore l'électronique.

1.2 Mitsubishi Electric ITE TCL

Mitsubishi Electric ITE TCL[1][2] est un laboratoire de recherches en télécommunication du groupe *Mitsubishi Electric* implanté à Rennes, en France. Créé en 1995, ce laboratoire se consacre à la conception d'outils de communication dans le domaine des télécoms sans fil et du réseau d'accès.

L'activité du laboratoire s'organise dans trois équipes distinctes :

- Équipe *télécoms et systèmes*, qui réalise des études et des développements sur les prochaines générations de systèmes de communication mobile, en particulier UMTS. Cette équipe travaille en particulier sur des outils de simulation dynamique, de planification de réseau et d'équipements de test et de validation ;
- Équipe *protocoles*, qui travaille sur des équipements destinés au réseau d'accès en domaine public ou en domaine privé. Le travail de cette équipe s'oriente désormais vers des équipements de cœur de réseau, avec un accent sur les aspects sécurité ;
- Équipe *communications numériques*, qui travaille sur la communication numérique (modulation, codage de canal, détection, égalisation), les protocoles (d'accès multiple, de gestion des erreurs, d'allocation de ressources) et les technologies de réalisation (architecture matérielle et logicielle, technologies associées ...) ;

Le laboratoire *Mitsubishi Electric ITE TCL* compte 35 salariés, dont 20 ingénieurs. Dans le cadre de mon stage, j'ai travaillé au sein de l'équipe *protocoles*, qui compte 13 ingénieurs.

[1] `http://www.mitsubishi-electric-itce.fr/`

[2] ITE = *Information TEchnology*, TCL = *TeleCommunication Lab*

Une partie de cette équipe travaille sur le projet *FlexNP*, sur lequel a porté mon stage, et qui est décrit en détail dans la partie suivante.

Chapitre 2

Le contexte du projet : plateforme *FlexNP*

2.1 Objectif

La plateforme *FlexNP* est une plateforme destinée au prototypage de fonctions réseau. Elle permet de prototyper des fonctionnalités réseau et de monitorer un réseau sur des densités de traffic de l'ordre du Gigabit/seconde. Cette plateforme est un projet de développement à long terme de l'équipe *Protocoles* du laboratoire. La conception et la réalisation de la plateforme matérielle a duré deux années, et l'équipe travaille actuellement au développement des applications.

Des exemples d'utilisation possibles sont :

- Interception d'un flux de vidéo-conférence et réservation de bande passante pour les flux vidéo et audio avant de garantir une qualité de service ;
- Interception de courriers électroniques et analyse anti-spam ;
- Interception de traffic réseau et détection d'intrusion ;

La plateforme *FlexNP* est prévue pour être souple et très configurable et donc s'adapter à diverses situations et applications. Il s'agit d'une plateforme de prototypage : il n'y a pas d'objectif de commercialisation, uniquement des objectifs de recherche et développement.

2.2 Fonctionnement

Elle est constituée de deux modules principaux : la **NPP**, le module de traitement et la **NAI**, le module d'accès. Chacun de ces modules prend la forme d'une carte au format *Compact PCI*. Ces cartes communiquent entre elles par l'intérmédiaire d'un bus *PCI* fond de panier dans un chassis (voir photo 2.3).

La **NAI** dispose d'interfaces réseau qui forment le lien avec le cœur du réseau, là où le traffic à analyser ou modifier arrive et part. La **NPP** dispose d'unités de traitements permettant d'effectuer ces analyses et ces modifications sur le traffic remonté par l'intermédiaire du module **NAI**. Mon stage n'a porté que sur le module *NPP*, c'est pourquoi on ne décrira dans la suite que ce module. La carte **NAI** était encore en phase de test matériel au moment de mon stage.

La carte *NPP* est constituée de trois parties :

- le *Control Plane Processor* (CPP) est un processeur MIPS double-cœur, chaque cœur opérant à une fréquence de 1 Ghz. Il permet d'effectuer des opérations avancées sur les paquets réseau, notamment au niveau de la couche application ;

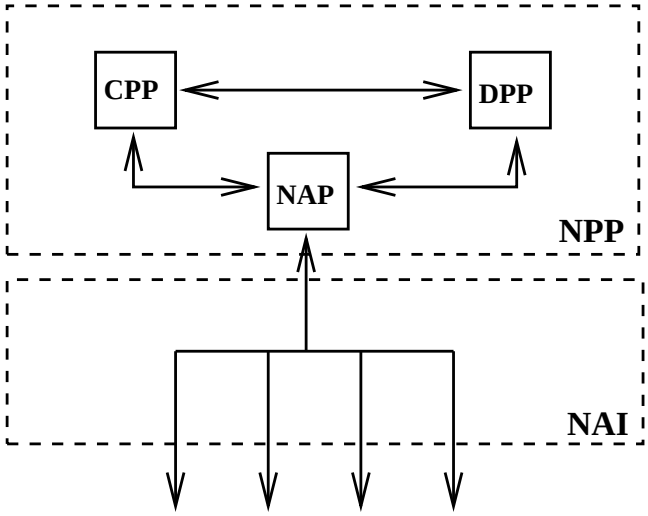

FIG. 2.1 – Schéma de structure NAI/NPP

- le *Data Plane Processor* (DPP) est un *network processor* de type Intel IXP 2400. Il permet de réaliser des opérations concernant les couches 2 à 4 de la pile de protocoles (filtrage IP, filtrage TCP, suivi de connexion, etc.) ;
- le *Network Access Processor* (NAP) est un FPGA Xilinx comportant plusieurs millions de portes logiques. Le *NAP* peut effectuer des opérations de base sur les paquets réseau reçus, uniquement au niveau des couches basses de la pile de protocoles. Il peut également servir à émuler une couche physique et une couche liaison pour réaliser du prototypage. Il est connecté au module *NAI* et peut transférer des données au *CPP* ou au *DPP*.

FIG. 2.2 – Photographie de la carte NPP

FIG. 2.3 – La carte NPP en fonctionnement dans un chassis

2.3 Partie CPP

2.3.1 Matériel

Le sujet de mon stage était centré sur la partie *CPP* de la *NPP*. Au cœur de la partie *CPP* se trouve un processeur d'architecture MIPS, le *RM9000*, commercialisé par la société *PMC-Sierra*[1]. Ce processeur possède deux cœurs fonctionnant chacun à la fréquence de 1 Ghz, et peut fonctionner soit en mode 32 bits, soit en mode 64 bits. Nous avons choisi de nous limiter à l'utilisation du mode 32 bits, qui est suffisant au regard des quantités de mémoire disponibles sur la plateforme. Ce processeur MIPS est directement connecté à une mémoire vive de 256 Mega-octets, grâce à un contrôleur mémoire intégré.

De plus, le processeur *MIPS* est connecté à un *chipset*, le *Marvell 64340*, comportant entre autre un contrôleur Ethernet Gigabit, un contrôleur de port série et un contrôleur mémoire. À ce *chipset* est connecté une petite mémoire rapide *SRAM* de 256 Kilo-octets ainsi qu'une mémoire vive plus lente de 128 Mega-octets.

La figure 2.4 détaille l'organisation de ces différents composants matériels.

2.3.2 Logiciel

Au début de mon stage, d'un point de vue logiciel, la partie *CPP* de la plateforme utilisait :

– le chargeur de système d'exploitation *PMON2000*. Ce chargeur est stocké en mémoire *ROM* et exécuté dès le démarrage de la plateforme. Il permet de charger le noyau d'un système d'exploitation et éventuellement un *ramdisk* puis de l'exécuter. Ce chargement peut avoir lieu à partir de la même mémoire *ROM* ou bien à partir du réseau, via le

[1] `http://www.pmc-sierra.com`

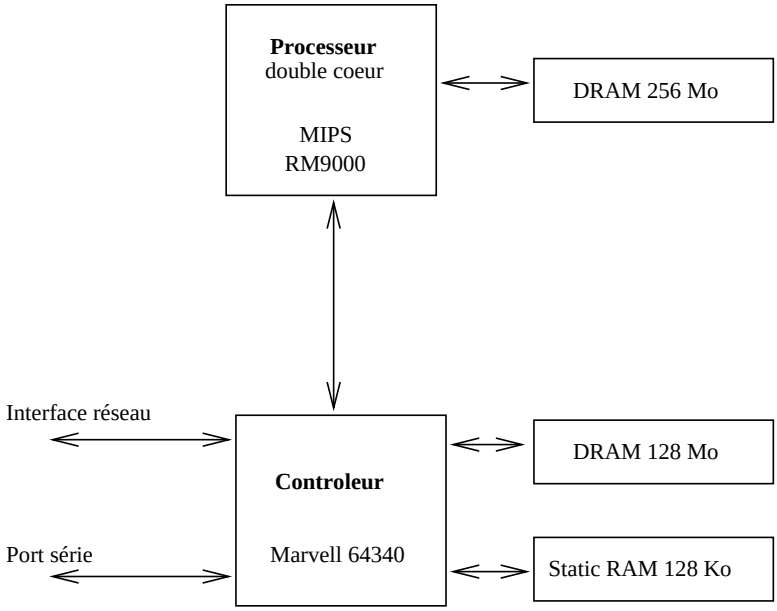

FIG. 2.4 – Schéma-bloc matériel de la partie *CPP*

protocole *TFTP*. C'est cette dernière méthode qui est utilisé en général pour les développements. Ce chargeur *PMON2000*, distribué sous une licence libre, a été modifié par l'équipe pour correspondre au fonctionnement de la plateforme ;
- un noyau Linux 2.4, modifié pour fonctionner sur la plateforme, voir partie 4 ;
- un *ramdisk* basé sur une RedHat, voir partie 3 ;

Dans le cadre de mon stage, j'ai travaillé successivement sur ces trois composants logiciels.

2.4 Difficultés

Cette plateforme possède donc une architecture particulière, très différente de l'architecture classique d'un ordinateur. La compréhension du fonctionnement général de la plateforme, du rôle des différents éléments et de leur organisation est au début du stage assez délicate.

Par ailleurs, l'utilisation de la plateforme et ses applications diffèrent grandement de celles d'un ordinateur classique. Il est donc nécessaire au début du stage de comprendre le contexte du projet, afin de mieux cerner les objectifs.

Le stage a donc comporté une première phase importante de prise de contact, de compréhension du contexte et des objectifs. Le travail effectif n'a donc pas commencé dès le début du stage, mais après une ou deux semaines de documentation et de discussion.

Chapitre 3

Chaînes de cross-compilation et *ramdisk*

La première partie du stage a consisté en la mise en place de chaînes de cross-compilation et l'automatisation de la génération de *ramdisk* pour la plateforme NPP.

3.1 Chaînes de cross-compilation

3.1.1 Problématique

Les outils de développement (compilateur et éditeurs de liens) disponibles sur les ordinateurs de développement sont prévus pour s'exécuter sur x86 et pour générer du code pour ce type de processeur. Ces outils permettent donc de compiler des applications qui vont s'exécuter sur l'ordinateur.

En revanche, la plateforme NPP utilise un processeur de type *MIPS*, donc le code généré par les outils de compilation disponibles par défaut sur les machines de développement ne convient pas. Il est nécessaire de générer du code spécifiquement pour l'architecture *MIPS*. En général, on ne souhaite pas compiler le code pour la plateforme embarquée directement sur la plateforme elle-même, car ses capacités mémoire ou de stockage sont insuffisantes ou tout simplement parce qu'il n'est pas commode de travailler dessus. On utilise donc une *chaîne de cross-compilation*, c'est-à-dire un ensemble d'outils de développements qui s'exécutent sur la machine de développement (à base de processeur x86) mais qui génère du code pour l'architecture *MIPS*.

GCC, *GNU Compiler Collection*[1] est un compilateur performant permettant de générer du code pour un grand nombre de processeurs. Il est très utilisé dans le cadre du développement pour l'embarqué. Il s'agit également d'un logiciel *libre* distribué sous les termes de la licence GPL.

Une *chaîne de cross-compilation* est constituée de trois éléments principaux :

- le compilateur lui-même, *GCC*, qui transforme du code C, C++ ou d'autres langages en code assembleur pour un processeur spécifique ;
- un ensemble d'outils de manipulation de binaires, les *binutils*[2]. Ils comportent notamment `as`, l'assembleur qui va transformer le code produit par *GCC* en un binaire utilisable sur la plateforme, `ld`, l'éditeur de liens et `objdump` qui permet d'observer et de désassembler un binaire ;

[1] `http://gcc.gnu.org`
[2] `http://www.gnu.org/software/binutils/`

– une bibliothèque standard C, comme la *GNU Libc*[3].

Au début de mon stage, l'équipe utilisait plusieurs chaînes de compilation, avec des versions de logiciels relativement anciennes, et parfois fournies par des entreprises externes. Elle souhaitait remplacer ces anciennes chaînes par de nouvelles et disposer de méthodes simples pour générer à nouveau ces *chaînes de cross-compilation*. La première étape de mon stage a consisté à répondre à ces besoins.

3.1.2 Choix des outils

Une première solution pour générer une chaîne de *cross-compilation* est de le faire manuellement, c'est à dire de compiler chaque outil individuellement, avec les options de configuration adaptées. En pratique, c'est un processus relativement complexe : certains outils nécessitent plusieurs passes de compilation, les options de configuration sont complexes et il faut trouver la combinaison des outils qui fonctionne correctement.

Bien que de nombreux documents expliquent le procédé permettant de mettre en place manuellement une chaîne de cross-compilation, il existe également plusieurs solutions automatisées simplifiant cette tâche.

J'ai tout d'abord étudié *Crosstool*[4], un ensemble de scripts shell qui automatise le téléchargement des outils, leur configuration, leur compilation et leur installation. Ils permettent de mettre en place relativement facilement une chaîne de cross-compilation. Toutefois, le mécanisme de configuration n'était pas très simple, les scripts shells peu lisibles et surtout, la seule bibliothèque standard C utilisable était la *GNU Libc*. Cette dernière, bien que parfaite pour une utilisation sur une machine de travail, est assez imposante en taille en raison de nombreuses fonctionnalités peu utiles dans le cadre d'une plateforme embarquée. Nous souhaitions plutôt utiliser la *uClibc*[5], une bibliothèque standard C de taille plus réduite mais qui offre quasiment toutes les fonctionnalités de la *GNU libc*.

Après avoir fait quelques essais en utilisant *Crosstool*, j'ai découvert *Buildroot*[6], un système automatisé de génération de chaînes de cross-compilation et de *ramdisk* (pour ce point, voir partie 3.2). Ce système est proposé par les développeurs de *uClibc* et supporte donc cette bibliothèque standard C. Il est basé sur un ensemble de *Makefile*, plus simples à modifier et à configurer que les scripts shells de *Crosstool*. Nous avons donc retenu *Buildroot* pour générer les chaînes de cross-compilation. Son utilisation est détaillée en section 3.3.

3.2 Ramdisk

La plateforme embarquée NPP utilise un noyau Linux pour fonctionner. Sur une machine classique, le noyau Linux utilise une partition du disque dur comme système de fichiers principal. La NPP ne disposant pas de disque dur, on utilise un *ramdisk*, c'est à dire l'image d'un système de fichiers. Celle-ci peut soit être stockée dans une mémoire *Flash* embarquée dans la plateforme, soit chargée par le réseau. C'est cette deuxième solution qui est utilisée pour tous les développements.

[3] `http://www.gnu.org/software/libc/libc.html`
[4] Logiciel Libre, sous licence GPL, `http://kegel.com/crosstool/`
[5] Logiciel Libre, sous la licence LGPL, `http://www.uclibc.org`
[6] Logiciel Libre, `http://www.uclibc.org/toolchains.html`

Le *ramdisk* contient donc l'image d'un système de fichiers complet, c'est à dire les applications, les bibliothèques partagées, les scripts d'initialisation, les répertoires temporaires, les pseudo-fichiers de périphériques (`/dev/`), etc...

Avant mon arrivée dans l'équipe, les *ramdisks* utilisés avaient été créés à partir de distribution GNU/Linux RedHat[7]. RedHat est une distribution classique prévue pour fonctionner sur un ordinateur de bureau, avec donc beaucoup de fonctionnalités inutiles pour une plateforme embarquée. J'ai donc été chargé de trouver un outil permettant de construire de manière automatisée un *ramdisk* minimal. Il s'est trouvé que *Buildroot* répondait également à ce besoin, et donc avec le même outil il était possible de générer à la fois la chaîne de cross-compilation et le *ramdisk*.

3.3 Buildroot

Buildroot est un jeu de Makefile automatisant le téléchargement, la configuration, la compilation et l'installation d'une chaîne de cross-compilation et d'un *ramdisk*.

Au début de mon stage, la configuration de *Buildroot* se faisait en éditant directement un *Makefile* principal. Il suffisait d'indiquer la version de *gcc*, de *binutils*, du *noyau* et les logiciels à installer dans le *ramdisk*. Au cours du stage, les développeurs de *Buildroot* ont amélioré leur système et y ont apporté un outil semi-graphique de configuration, comme celui que l'on peut trouver pour la configuration du noyau Linux (`makemenuconfig`).

Cet outil de configuration (voir figure 3.1) permet en particulier de sélectionner :

- les options de la chaîne de cross-compilation, en particulier les versions des outils de développement à utiliser ;
- les logiciels à inclure dans le *ramdisk*. En particulier, pour toutes les commandes standard comme `ls`, `cp`, etc., *Buildroot* utilise *Busybox*[8] ce qui permet de singulièrement réduire la taille occupée dans le *ramdisk*. De plus, toutes les applications sont liées avec la bibliothèque standard C *uClibc*, beaucoup plus petite que la *GNU Libc* ;
- le format du système de fichiers à utiliser pour le *ramdisk* ;

J'ai été amené à réaliser diverses améliorations à *Buildroot* dont la plupart ont été intégrées à la version officielle du logiciel. Ces améliorations permettent d'accélérer la vitesse de compilation sur une machine multi-processeur (utilisation de l'option `-j` de `make`), de générer la chaîne de cross-compilation à un emplacement personnalisable et non plus fixe et d'installer automatiquement la bibliothèque standard C++ dans le *ramdisk*. J'ai également rédigé une documentation au format *HTML*, qui a été intégrée au projet *Buildroot*, dans le répertoire `doc`. Celle-ci détaille de manière assez précise le fonctionnement interne du système, comment ajouter de nouveaux composants ou comment modifier la configuration des logiciels intégrés au *ramdisk*. Elle est disponible en annexe.

En plus de ces modifications de *Buildroot*, j'ai mis en place un squelette de *ramdisk* utilisé par les *Makefile*. Ce squelette contient les scripts d'initialisation personnalisés pour la plateforme NPP. Pour ces scripts, j'ai été amené à écrire un petit programme C qui récupère l'adresse IP de la plateforme dans la mémoire *Flash* où elle est stockée. Cela permet à chaque plateforme de configurer automatiquement son interface réseau avec l'adresse IP qui lui a été attribuée.

Finalement, *Buildroot* a permis de générer des chaînes de cross-compilation plus récentes,

[7] `http://www.redhat.com`
[8] Logiciel Libre, sous licence GPL, `http://www.busybox.net/`

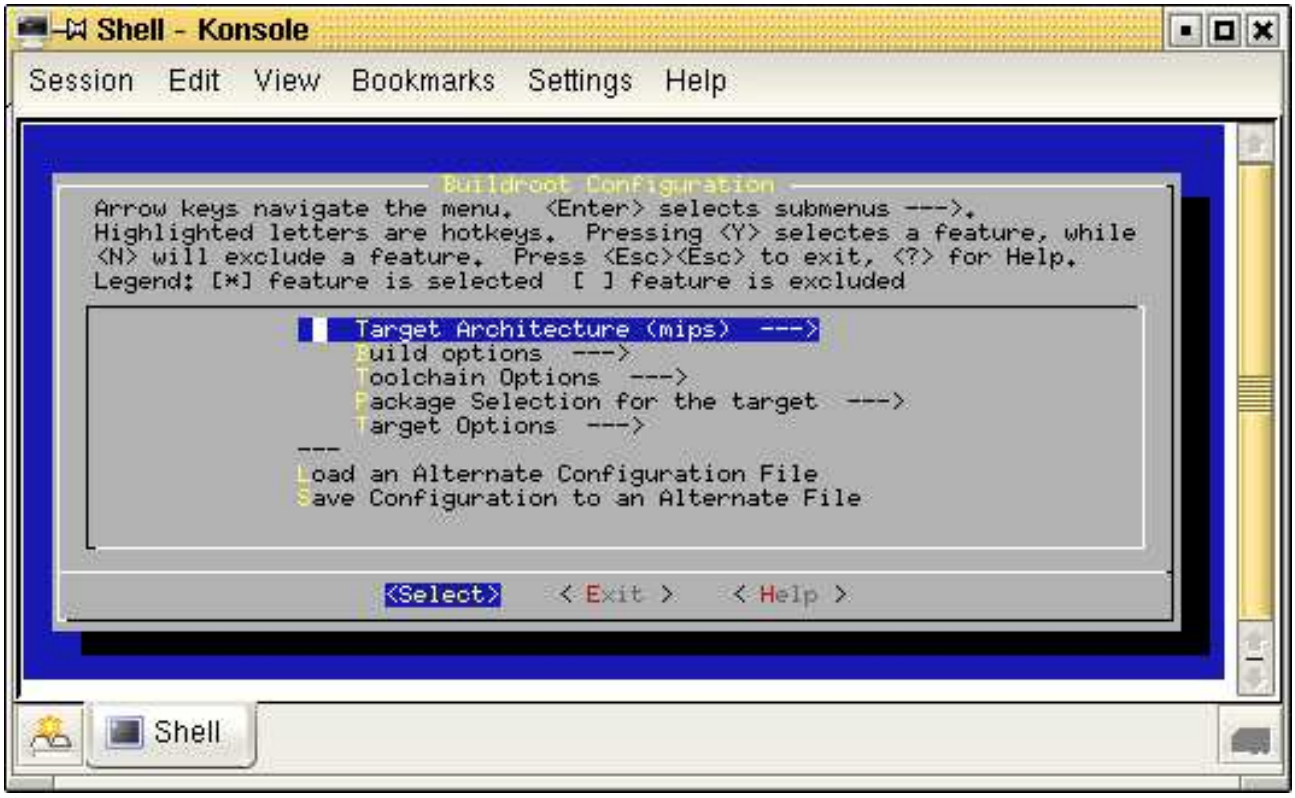

FIG. 3.1 – Outil de configuration de *Buildroot*

en utilisant notamment la version 3.4.2 de *GCC*, disposant d'optimisations spécifiques pour le processeur *MIPS RM9000*. L'utilisation d'*uClibc* a permis de réduire le temps nécessaire à la génération de la chaîne de cross-compilation. L'aspect minimal du *ramdisk* a permis d'obtenir un système de fichiers compressé d'environ 900 Ko contenant toutes les applications nécessaires, y compris la bibliothèque standard C++. Pour des fonctionnalités similaires, l'ancien *ramdisk* occupait 17 Mo, était donc plus long à charger et plus difficile à maintenir. *Buildroot* permettra également d'ajouter de manière très simple de nouveaux logiciels dans le *ramdisk* si cela s'avère nécessaire.

Initialement destiné à la génération de chaîne de cross-compilation et de *ramdisk* pour la partie MIPS de la plateforme, nous avons finalement utilise également *Buildroot* pour la partie ARM de la NPP. Les deux chaînes de cross-compilation et les deux *ramdisks* sont donc générés avec exactement la même configuration, ce qui facilite grandement leur maintenance.

Ces deux chaînes de compilation et ces deux *ramdisks* ont été livrés à d'autres laboratoires de recherche japonais de *Mitsubishi Electric* et sont donc utilisés aujourd'hui pour tous les développements sur la plateforme *NPP*.

3.4 Conclusion

L'outil *Buildroot* répond maintenant bien aux besoins de l'équipe *Protocoles* pour la génération de chaînes de cross-compilation et de *ramdisks* personnalisés.

La sélection des outils et leur mise en place a occupé environ le premier mois du stage, mais il a été nécessaire tout au long du stage d'améliorer les *ramdisks* en fonction des besoins.

Chapitre 4

Portage du noyau Linux 2.6

4.1 Motivations

Avant mon arrivée, l'équipe utilisait pour la partie *MIPS* de la plateforme NPP un noyau Linux 2.4. Plus exactement, il s'agissait d'un noyau 2.4.19-rc1 livré avec une carte d'évaluation d'une plateforme assez similaire à la plateforme NPP. Ce noyau avait été modifié pour correspondre à la configuration matérielle de la NPP.

L'équipe souhaitait que je porte le noyau Linux 2.6 pour plusieurs raisons. Tout d'abord, ce dernier offre de plus nombreux pilotes de périphériques, notamment en ce qui concerne le support des mémoires *Flash* et du système de fichiers *Jffs2*. De plus, nous ne savions pas encore à ce moment quel mécanisme utiliser pour le fonctionnement multi-processeur de la plateforme. Il nous a donc paru pertinent de partir sur un noyau supportant correctement d'office le multiprocesseur. Le noyau 2.4 le supporte également, mais beaucoup d'améliorations ont été apportées avec le 2.6 à ce sujet.

J'ai donc été chargé de porter le noyau Linux 2.6 sur la plateforme, sachant qu'un noyau 2.4 existait déjà. Une base de code existante était donc disponible, il fallait l'adapter pour le noyau 2.6. Toutefois, le noyau 2.6 constitue une évolution majeure par rapport au noyau 2.4, le fonctionnement et la structuration du code a donc beaucoup changé et les adaptations nécessaires ne sont pas nécessairement triviales.

4.2 Documentation

Avant de commencer le portage du noyau Linux proprement dit, il a fallu que je me documente sur divers aspects matériels et logiciels. Pour cela, j'ai parcouru plusieurs documents, que j'ai ensuite utilisé tout au long de mon stage :

- *See Mips Run*, de Dominique Sweetman, un ouvrage général sur l'architecture *MIPS*, 480 pages ;
- *RM92000A User Manual*, de PMC-Sierra, la *datasheet* du processeur, 660 pages ;
- *MV64340, System controller for MIPS processors*, de Marvell Solutions, la *datasheet* du chipset, 690 pages ;
- *Understanding the Linux Kernel*, de Daniel Bovet et Marco Cesati, 780 pages ;
- *Linux Device Drivers*, de Alessandro Rubini et Jonathan Corbet, 580 pages

Ces documents représentent une masse d'information très importante, difficile à assimiler. En particulier, les *datasheets* sont des documents assez difficiles d'accès. La recherche

d'informations et la compréhension de l'architecture a représenté, pendant toute la durée du stage, une importante partie du travail.

4.3 Gestion de la mémoire sur MIPS

Afin de comprendre le fonctionnement du noyau Linux sur la plateforme NPP ainsi que le fonctionnement du multi-processeur, il est nécessaire de donner quelques détails sur la gestion de la mémoire sur architecture *MIPS*. La figure 4.1 décrit l'organisation de l'espace d'adressage du MIPS.

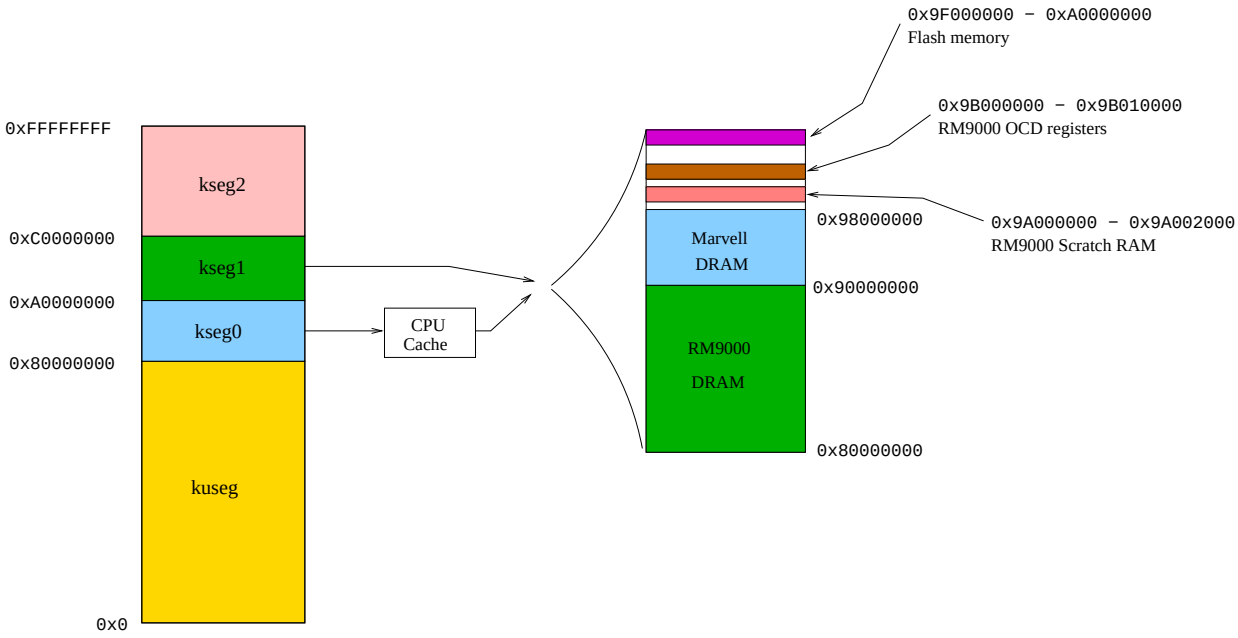

FIG. 4.1 – Organisation d'un espace d'adressage sur processeur MIPS

En mode 32 bits, l'espace d'adressage du MIPS a une taille de 4 Giga-octets et est découpé en quatre parties distinctes :

- `kuseg` est une zone de 2 Go utilisée pour *mapper* le code et les données des applications utilisateur. Les accès à cette zone sont donc possibles depuis du code non privilégié, par exemple depuis des applications utilisateur. Lors d'un accès à cette zone, l'adresse virtuelle est convertie en une adresse physique par l'unité de gestion de mémoire du processeur (*MMU*) ;
- `kseg0` est une zone de 512 Mo utilisée pour *mapper* à l'identique les 512 premiers Mega-octets de la mémoire physique et accessible uniquement en mode noyau. Les accès mémoire effectués par cette zone ne sont donc pas traduits par la *MMU*, mais passent par le cache du processeur ;
- `kseg1` est une zone de 512 Mo utilisée pour *mapper* à l'identique les 512 premiers Mega-octets de la mémoire physique. Son contenu est donc identique à celui de `kseg0`, sauf que les accès ne passent pas par le cache du processeur, ils sont effectués directement dans la mémoire ;
- `kseg2` est une zone de 1 Go accessible uniquement en mode noyau. Tous les accès effectués sur cette zone sont traduits par la *MMU*.

La mémoire physique elle-même est constituée de plusieurs éléments, qui seront donc accessibles au travers de `kseg0` et `kseg1`. Les adresses indiquées sur la droite de la figure 4.1 sont celles utilisées lorsqu'on accède aux données au travers de `kseg0`. Pour un accès au travers de `kseg1` aux mêmes données, les adresses sont identiques, à un offset de `0x20000000` près.

La mémoire physique se décompose en 5 éléments :

- La *DRAM* directement attachée au processeur RM9000, d'une taille de 256 Mo, située au début de la mémoire physique ;
- La *DRAM* attachée au *chipset Marvell MV64340*, d'une taille de 128 Mo, située juste après la *DRAM* du RM9000 ;
- La *Scratch RAM* du processeur est une petite mémoire de 8 Ko interne au processeur *RM9000* ;
- La zone *OCD registers* est une zone contenant des registres de configuration pour le processeur *RM9000* ;
- La *Flash* est une mémoire EEPROM contenant le chargeur de système d'exploitation de la plateforme ;

4.4 Noyau utilisé

Plutôt que d'utiliser un noyau Linux 2.6 officiel, j'ai préféré m'orienter avec un noyau Linux fourni par les mainteneurs du port *MIPS* du noyau. En effet, le noyau qu'ils fournissent est bien plus à jour et bien plus évolué en ce qui concerne le support de l'architecture MIPS. De plus, ce noyau Linux-MIPS est régulièrement synchronisé avec le noyau Linux officiel, ce qui permet de bénéficier de toutes les améliorations de ce dernier pour tout ce qui est extérieur au support de l'architecture *MIPS*.

La communauté des développeurs du noyau Linux-MIPS est organisée autour d'un site Web, `http://www.linux-mips.org`, d'une liste de diffusion et d'un serveur CVS. Dès le début du stage, je me suis inscrit à la liste de diffusion afin de prendre contact avec les développeurs dont les conseils et explications m'ont été fort utiles.

J'ai donc commencé par récupérer un noyau 2.6 à partir du serveur CVS de Linux-MIPS[1]. Tout au long du stage, j'ai régulièrement synchronisé cette version avec les nouvelles améliorations des développeurs de Linux-MIPS. Cela nécessitait de fusionner les modifications que j'avais effectuées avec celles effectuées par les développeurs Linux-MIPS. Ce travail est parfois un peu fastidieux, mais nécessaire pour bénéficier des dernières améliorations. Malheureusement, le travail que je réalisais sur le noyau Linux n'était pas intégrable au CVS : il concernait le support d'une architecture matérielle très particulière que personne ne peut acquérir. C'est la raison pour laquelle, bien que j'étais tout à fait autorisé voire encouragé à collaborer avec les développeurs de Linux-MIPS dans l'esprit du Logiciel Libre, l'intégration de mon code ne présentait aucun intérêt.

Dans toute la démarche de portage du noyau 2.6, j'ai utilisé diverses documentations, en particulier le *Linux MIPS Porting Guide*[2] de Jun Sun, un développeur de l'entreprise Montavista. Au cours de mon stage, cette documentation a été intégrée au *Wiki*[3] du site de Linux-MIPS.

[1] `http://www.linux-mips.org/wiki/index.php/Net_Resources\#Anonymous_CVS`
[2] `http://linux.junsun.net/porting-howto/porting-howto.html`
[3] `http://www.linux-mips.org/wiki/index.php/Porting`

4.5 Initialisation de la plateforme

4.5.1 Séparation du code

Dans un noyau de système d'exploitation qui a vocation à être portable, c'est à dire disponible sur plusieurs architectures de processeur, on sépare en général le code en deux parties. La première partie, la plus importante en taille est la partie *indépendante* de l'architecture, qui fonctionnera donc quel que soit le processeur. La seconde partie, en général plus réduite est la partie *dépendante* de l'architecture.

Le noyau GNU/Linux est bien évidemment portable, disponible pour l'architecture *x86*, mais également pour une vingtaine d'autres architectures, telles que *alpha*, *arm*, *cris*, *ia64*, *m32r*, *m68k*, *ppc*, *sparc*, et bien sûr *mips*.

Pour l'architecture *MIPS*, le code dépendant de l'architecture est regroupé dans le répertoire `arch/mips/`. Les entêtes dépendant de l'architecture sont dans le répertoire `include/asm-mips/`, `include/asm/` devenant un lien symbolique vers ce répertoire lors de la compilation.

Toutefois, la séparation en différentes architectures du code source n'est pas suffisante. En effet, différentes *plateformes* peuvent utiliser une même architecture. Dans le cadre de l'architecture *x86*, ce n'est pas une distinction primordiale car la plateforme *PC* classique est quasiment la seule plateforme à faire usage de ce type de processeurs. En revanche, dans le cadre d'autres architectures, en particulier des architectures utilisées dans l'embarqué tel que *ARM*, *PowerPC* ou *MIPS*, il existe une multitude de plateformes.

Les plateformes *MIPS* utilisent toutes un processeur *MIPS*, mais diffèrent de par leur configuration de la mémoire physique, de leur processus de démarrage, leur configuration matérielle, leur périphériques d'entrées-sorties, etc.

Ainsi, au sein du répertoire `arch/mips/` dédié au support de l'architecture *MIPS*, on trouve un certain nombre de répertoires. Certains d'entre eux, comme `mm`, `kernel`, `boot` ou `lib` contiennent du code commun à toutes les plateformes *MIPS*. Les autres répertoires contiennent du code spécifique à une plateforme donnée, comme par exemple `au1000`, `dec` ou `momentum`.

Pour le code spécifique à la plateforme NPP a donc été créé un répertoire `npp-board`.

4.5.2 Configuration

Pour permettre la sélection de la plateforme par l'intermédiaire de l'outil de configuration du noyau (`makemenuconfig` ou `makexconfig`), une entrée a été ajoutée pour la plateforme NPP dans le fichier `arch/mips/Kconfig` :

```
config NPP_BOARD
        bool "Support for Mitsubishi NPP board"
        select DMA_NONCOHERENT
        select HW_HAS_PCI
        select IRQ_CPU
        select IRQ_CPU_RM7K
        select IRQ_MV64340
        select PCI_MARVELL
        select RM7000_CPU_SCACHE
        select SWAP_IO_SPACE
```

```
        help
          The NPP board is MIPS-based development board made by
          Mitsubishi Electric ITE-TCL.
```

La sélection de la plateforme NPP entraîne donc la sélection de différents paramètres du noyau relatifs à la gestion des IRQs, du PCI ou des DMAs.

En plus de cela, d'autres paramètres ont été ajoutés au fichier `arch/mips/Makefile` :

```
#
# Mitsubishi NPP board
#
core-$(CONFIG_NPP_BOARD)   += arch/mips/npp-board/
cflags-$(CONFIG_NPP_BOARD) += -Iinclude/asm-mips/mach-npp
load-$(CONFIG_NPP_BOARD)   += 0x80200000
```

Ces paramètres permettent d'inclure d'autres répertoires dans les chemins par défaut des entêtes du noyau et de fixer l'adresse de chargement du noyau dans la mémoire.

4.5.3 Description du code

Le code spécifique à la plateforme NPP se trouve donc dans le répertoire `arch/mips/npp-board`. Il contient plusieurs fichiers dont nous décrivons le contenu ci-dessous. Ce code est inspiré du code de la plateforme *Jaguar ATX* que l'on peut trouver dans le répertoire `arch/mips/momentum/jaguar_atx`.

- `dma-alloc.c` : code spécifique utilisé pour le pilote *Ethernet*, décrit en section 4.7 ;
- `int-handler.S` : code des gestionnaires d'interruption de premier niveau, en assembleur ;
- `irq.c` : code de gestion des IRQs ;
- `irq-cic.c` : code de gestion du *Central Interrupt Controller* du processeur *RM9000* ;
- `mv64340-eth-fix.c` : code spécifique utilisé pour le pilote *Ethernet*, décrit en section 4.7 ;
- `prom.c` : code permettant de récupérer et d'analyser des paramètres passés par le chargeur du noyau. Les fonctions présentes sont indispensables pour compiler le noyau, mais nous n'utilisons pas ces fonctionnalités ;
- `reset.c` : code permettant d'effectuer un redémarrage de la plateforme. Les fonctions présentes sont indispensables pour pouvoir compiler le noyau, mais pour des questions de configuration mémoire, il n'est pas possible de redémarrer la plateforme par logiciel ;
- `setup.c` : code d'initialisation de la plateforme proprement dit ;

Les fichiers d'en-têtes sont répartis dans différents répertoires :

- `asm/marvell.h` est présent à l'origine dans le noyau Linux et contient des macros pour accéder aux registres du chipset *Marvell 64340* ;
- `linux/mv643xx.h` est présent à l'origine dans le noyau Linux et contient des définitions de constantes pour la programmation du chipset *Marvell MV64340* ;
- `asm/mach-npp/board.h` a été créé spéciquement pour la plateforme. Il contient les définitions propres à la carte *NPP* : tailles et adresses des différentes mémoires, définition des plages d'interruption ;

- `asm/mach-npp/cpu-feature-overrides.h` a été créé spécifiquement pour la plateforme mais a un format commun a toutes les plateformes. Il permet de définir des macros qui modifient le comportement du code générique de l'architecture *MIPS*, en ce qui concerne les caches, l'émulation des instructions, etc. ;
- `asm/mach-npp/rm9000.h` a été créé spécifiquement pour la plateforme. Il contient des définitions propres au processeur *RM9000* ;

Fichier `Setup.c`

Le fichier `setup.c` contient le code d'initialisation de la plateforme NPP. Plus précisement, c'est la fonction `npp_board_setup` qui se charge de cette initialisation. L'appel à la macro `early_initcall` à la fin du fichier permet de déclarer cette fonction comme étant une fonction d'initialisation à appeler dès le début de l'exécution du noyau.

C'est l'appel à la fonction `do_earlyinitcalls` dans la fonction `setup_arch` de `arch/mips/kernel/setup.c` qui déclenche l'appel à `npp_board_setup`.

Cette dernière se charge des opérations suivantes :

- définir des pointeurs de fonctions indispensables au fonctionnement du noyau, comme les fonctions pour la gestion du temps (`board_time_init`, `board_timer_setup`) et pour le redémarrage de la machine (`_machine_restart` ...) ;
- initialise des entrées de TLB (*Translation Lookaside Buffers*) fixes pour permettre l'accès à la mémoire interne du contrôleur mémoire, réseau et série ainsi qu'à ses registres de configuration ;
- stoppe les contrôleurs DMA de la ligne série ainsi que les contrôleurs Ethernet ;
- déclare la mémoire physique disponible pour le noyau Linux, par l'appel à la fonction `add_memory_region`. Le noyau Linux fonctionne dans le segment *kseg0* de l'espace d'adressage du *MIPS*. Ce segment est une représentation à l'identique de la mémoire physique. Toutefois, on ne déclare pas toute la mémoire physique comme étant accessible par le noyau Linux, on le limite aux 256 Mo de mémoire vive attachés directement au processeur *MIPS*. En réalité, sur ces 256 Mo, on réserve encore un peu de mémoire pour le fonctionnement multi-processeur, voir partie 5 ;
- appelle les fonctions d'initialisation pour les fonctionnalités présentes dans les fichiers `mv64340-eth-fix.c` et `dma-alloc.c`, voir section 4.7.

Gestion des interruptions

La gestion des interruptions au niveau *MIPS* sur la plateforme NPP est très complexe et sa compréhension a été relativement longue. Le code nécessaire à leur gestion est réparti dans les fichiers `int-handler.S`, `irq.c` et `irq-cic.c`. Au cours du stage, j'ai rédigé une courte documentation en anglais au sujet du fonctionnement des interruptions, elle est disponible en annexe.

4.6 Pilote série

La plateforme *NPP* ne dispose pas d'un contrôleur série standard de type *16550* comme l'on en trouve dans les PCs et sur de nombreuses cartes de développement. Il est donc nécessaire d'utiliser le contrôleur série fourni par le chipset *Marvell 64340*. Malheureusement, le

noyau Linux ne propose pas de support pour ce type de port de série. Il a donc été nécessaire d'écrire un pilote de périphérique série spécifique pour ce chipset.

Un pilote de périphérique série avait déjà été développé par l'équipe sur le noyau 2.4. Toutefois, ce dernier utilisait directement l'infrastructure des périphériques caractères, ce qui rendait son code relativement long et complexe. Dans le cadre du développement du nouveau driver sur le noyau 2.6, j'ai choisi d'utiliser la nouvelle infrastructure permettant d'implémenter de manière simple des pilotes série. Cette infrastructure est décrite de manière brève dans un article du *Linux Journal*, *Tiny TTY driver*[4] ainsi que dans un petit document livré avec le noyau Linux[5]. Les documentations étant relativement peu nombreuses sur le sujet, j'ai également utilisé les autres pilotes de périphériques série utilisant cette infrastructure comme modèle.

Le pilote se décompose en deux parties : une partie simple permettant uniquement d'effectuer des sorties de caractères, qui est utilisée pour la console noyau, et une partie plus complexe, gérant réception et émission, utilisée par toutes les applications utilisateur. Ces deux parties sont décrites dans les deux sections suivantes.

4.6.1 Console noyau

Le pilote série de la console noyau est implémenté dans le fichier `drivers/serial/mv64340-console.c`. Ce pilote permet uniquement d'écrire des caractères sur le port série. Il n'utilise donc pas les interruptions ni le DMA. La fonction `serial_outc` est chargée d'écrire un caractère sur le port série, en utilisant deux fonctions internes `write_char` (écriture du caractère proprement dite), `wait_for_xmitr` (attente de la fin de la transmission).

À partir de `serial_outc`, la fonction `mv64340_serial_write` permet d'écrire une chaîne complète sur le port série. La fonction `mv64340_serial_setup` est théoriquement chargée de configurer le contrôleur série. Dans notre cas, cette dernière ne fait rien car la configuration a déjà été effectuée par le chargeur du noyau, le *PMON*.

La structure `sercons` définit les divers paramètres et méthodes de la console série, et permet à la fonction `mv64340_console_init` d'enregistrer la console par un appel à `register_console`.

Ce pilote simple est donc utilisé par le noyau pour afficher les messages utilisant la fonction `printk`. Les applications utilisateur disposent d'un mécanisme plus évolué de terminaux, qui nécessite un pilote plus complet, décrit dans la section suivante.

4.6.2 Pilote complet

Le pilote série complet est présent dans le fichier `drivers/serial/mv64340.c`. L'objet du pilote est d'implémenter un ensemble de fonctions que la couche *serial core* utilisera pour proposer un périphérique de type `/dev/ttyS0` au système. Ces fonctions sont listées dans la structure `mv64340_ops`.

Principe

Pour l'implémentation de ce pilote, le mode *DMA* est utilisé. Il aurait été plus simple d'utiliser le mode *Debug* du contrôleur, mais malheureusement, celui-ci ne fonctionne pas

[4] `http://www.linuxjournal.com/article.php?sid=6331`
[5] Documentation/serial/driver

pour la réception en raison d'un bug du matériel. Ce problème avait déjà été soulevé lors de l'écriture du pilote dans le noyau 2.4. Toutefois, alors que le noyau 2.4 utilisait le mode *Debug* pour l'émission, le pilote réalisé pour le 2.6 utilise le mode *DMA* à la fois pour l'émission et la réception.

En mode *DMA*, le contrôleur série du *Marvell 64340* fonctionne de manière assez similaire à un contrôleur Ethernet. Le pilote doit construire et gérer des chaînes de *buffers* DMA.

Pour la réception, on indique au contrôleur le début de la chaîne de *buffers* DMA. Lorsque des informations arrivent sur le port série, les *buffers* DMA seront remplis au fur et à mesure sans nécessiter l'intervention du processeur. Quand le contrôleur le décidera, il avertira alors le processeur de la disponibilité des données à l'aide d'une interruption. Le pilote de périphérique récupèrera ces données dans les *buffers* DMA, les transfèrera aux couches plus hautes du système d'exploitation et replacera les *buffers* DMA dans la chaîne pour la future réception.

En émission, le système d'exploitation envoie des données au pilote de périphérique, qui remplit des *buffers* DMA au fur et à mesure. A chaque fois que cela est nécessaire, le pilote avertit le contrôleur qu'un *buffer* DMA contient des données à envoyer.

Implémentation

Localisation des buffers DMA Les *buffers* DMA sont alloués dans la petite *Static RAM* de 256 Ko du chipset. Ils ne peuvent être alloués dans la mémoire principale attachée au processeur, car elle n'est pas visible depuis le *chipset*. Pourtant, puisque l'on utilise le mode *DMA*, ce dernier a besoin d'accéder directement aux buffers. Il aurait été possible de les allouer dans la mémoire vive de 128 Mo attachée au chipset, mais étant donné la petite taille des buffers, il a été choisi pour l'instant de les placer dans la *Static RAM*.

Le fichier `include/asm/mv64340/mv64340_uart.h` contient les définitions liées au contrôleur série. En particulier, `SDMA_UART_BASE_ADDR` désigne l'adresse du début de la zone réservée dans la *Static RAM* pour les *buffers* DMA. Pour la réception, `SDMA_RX_BUFFER_NUM` (8 par défaut) *buffers* sont alloués, et pour l'émission, ce sont `SDMA_TX_BUFFER_NUM` (1 par défaut) qui sont alloués.

En réalité, à chaque *buffer* DMA, d'une taille de 256 octets (voir `SDMA_xx_BUFFER_SIZE`), est associé un descripteur d'une taille de 16 octets. Ce sont ces descripteurs, chaînés entre eux, qui forment la chaîne de buffers DMA, comme le montre la figure 4.2.

Initialisation L'initialisation du pilote est réalisée par la fonction `mv64340_init`. Elle consiste à enregistrer le pilote lui-même, puis le port série, et enfin l'IRQ (interruption) utilisée pour l'acquitemment des transmissions et le signalement des réceptions. Enfin, les fonctions `initialize_sdma_rx_buffers` et `initialize_sdma_tx_buffers` initialisent les *buffers* DMA et les chaînes de descripteurs associés.

Réception Lors de la réception d'un ou plusieurs caractères sur le port série, le contrôleur remplit un ou plusieurs *buffers* DMA puis signale cet évènement au processeur à l'aide d'une interruption. Le gestionnaire de l'interruption enregistrée précédemment, `mv64340_interrupt`, prend alors la main. Dans le cas où il s'agit d'une réception, une fonction auxiliaire, `mv64340_do_rx` est appelée.

Cette dernière parcourt l'ensemble des *buffers* DMA reçus, et pour chacun d'entre eux, extrait les données qui y sont contenues et les transmet aux couches supérieures du noyau

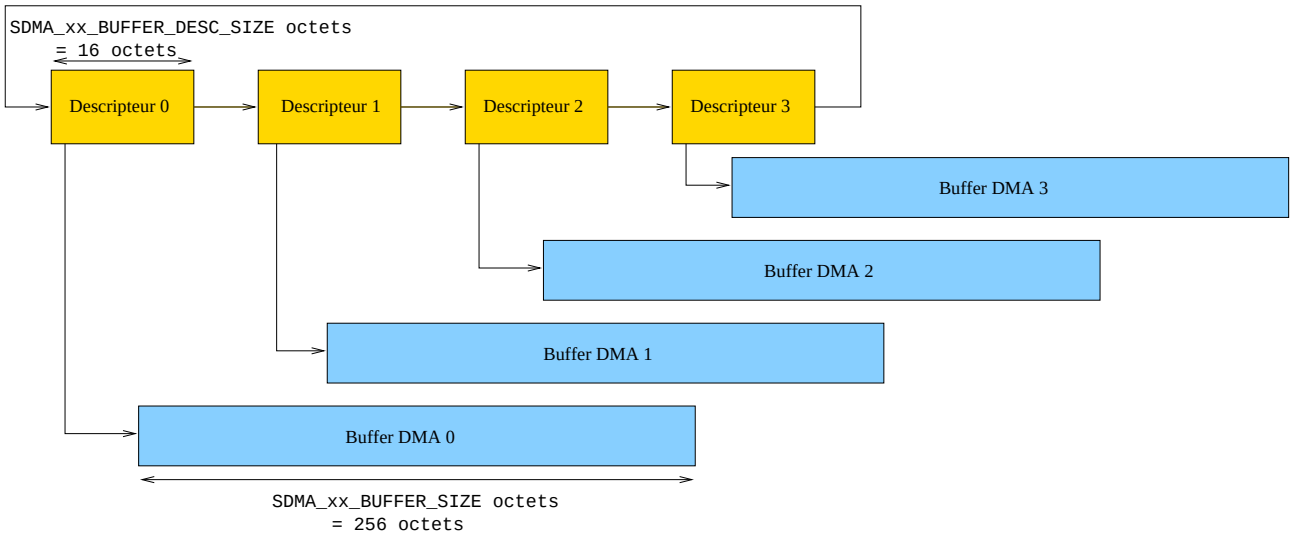

FIG. 4.2 – Fonctionnement des *descripteurs* DMA et des *buffers* DMA

chargées de la gestion des terminaux. Ceci est réalisé dans la fonction `mv64340_do_rx_buffer`.

Au fur et à mesure de leur consommation, les *buffers* DMA sont replacés dans l'état disponible pour le contrôleur série, de manière à ce que de nouvelles réceptions puissent avoir lieu.

Émission Pour l'émission, les couches hautes du noyau de gestion des terminaux n'appellent pas directement une fonction du pilote de périphérique série. Elles activent simplement les interruptions en transmission par un appel à `mv64340_start_tx`. C'est lors de la réception de cette interruption que la transmission a lieu, acquittement après acquittement.

Pour que ce mécanisme fonctionne, la fonction `mv64340_start_tx`, en plus d'activer les interruptions de transmission, démarre une transmission factice. Cette transmission ne transmet aucun caractère, mais permet de déclencher une interruption d'acquittement de fin de transmission, ce qui va enclencher le processus. Le gestionnaire d'interruption `mv64340_interrupt` est alors appelé. Dans le cas d'une transmission, celui-ci appelle la fonction `mv64340_tx_chars` qui va ajouter un par un les caractères du buffer d'attente des couches hautes au buffer DMA de transmission, avant de déclencher à nouveau le processus au niveau du contrôleur.

Lorsque la transmission est complètement terminée, après éventuellement plusieurs interruptions si la quantité de données à transférer est importante, les couches supérieures appellent `mv64340_stop_tx` pour désactiver les interruptions de transmission, et donc stopper le processus.

4.6.3 Conclusion et améliorations possibles

Bien que disposant d'un pilote pré-existant pour le noyau 2.4, ce pilote série pour le noyau 2.6 a été relativement long à écrire. Tout d'abord, il a fallu comprendre le fonctionnement de l'infrastructure *serial-core*, puis le fonctionnement assez complexe du contrôleur série. D'autre part, le déboguage d'un tel code est assez délicat puisqu'il est en contact direct avec le matériel. Lorsque quelque chose ne fonctionne pas, on ne dispose que de très peu

d'informations permettant de diagnostiquer l'origine du problème.

Ce pilote de périphérique série est améliorable sur plusieurs points :

- la transmission d'information de manière très rapide (par exemple en copiant/collant un gros morceau de texte) entraîne parfois un gel du pilote en transmission. La plateforme est toujours active, joignable par le réseau, mais la transmission série ne fonctionne plus. Après diverses investigations, j'ai suspecté un problème matériel, notamment parce qu'un *Errata* concernant une *race condition* lors de l'utilisation de la transmission et du DMA a été publiée par le constructeur *Marvell* ;
- les buffers sont actuellement stockés en *Static RAM*. Il serait intéressant de les déplacer dans la *DRAM* de 128 Mo attachée au *Marvell*. Cela permettrait de libérer de la place dans la *Static RAM*, de taille très petite, pour d'autres applications ;

4.7 Pilote Ethernet

Le contrôleur Ethernet Gigabit, comme le contrôleur série, est fourni par le chipset *Marvell MV64340*. Sur la carte NPP, une interface réseau est connectée à ce contrôleur. Cette interface réseau n'est pas celle utilisée pour le chemin de données à très haut débit traversant la plateforme. Elle est uniquement utilisée à des fins de déboguage et de monitoring distant. Les performances de son pilote ne sont donc pas critiques.

Dans le noyau Linux 2.6, un pilote Ethernet existe déjà pour ce chipset. Il est situé dans le fichier `drivers/net/mv643xx_eth.c`. Il n'était donc pas nécessaire d'écrire un pilote, mais des adaptations étaient nécessaires.

4.7.1 *Socket buffers* et DMA : problématique

Dans le noyau Linux, les paquets réseau sont stockées dans des *socket buffers*, dont le nom est souvent abrégé en `skb`. Ainsi, lors de l'émission d'un paquet, un `skb` est alloué lors de l'entrée du paquet dans la pile de protocles. Ce `skb` est utilisé dans toute la pile réseau pour ajouter l'encapsulation des différents protocoles avant d'être transféré au pilote de la carte réseau. De la même façon, lorsqu'un paquet est reçu sur la carte réseau, le pilote le place dans un `skb` qu'il envoie dans la pile réseau. Ce `skb` est alors désencapsulé en remontant les couches de protocoles de la pile réseau.

En général, les contrôleurs Ethernet fonctionnent en *DMA*, ce qui permet de libérer le processeur lors du transfert d'un paquet réseau depuis la mémoire vers le contrôleur ou depuis le contrôleur vers la mémoire. Pour cela, il faut que les *buffers DMA* soient accessibles depuis le contrôleur Ethernet.

Dans le pilote réseau pour le *Marvell MV64340* fourni dans le noyau Linux, les *sockets buffers* sont directement utilisés comme les *buffers DMA*. C'est en effet la meilleure stratégie, car cela évite des recopies inutiles de données en mémoire. Les *sockets buffers* sont alloués et libérés par des fonctions telles que `skb_alloc` et `skb_free` du fichier `net/core/skbuff.c`. Ils sont alloués dans la mémoire gérée par le noyau Linux, c'est à dire dans le cas de la plateforme NPP, dans la *DRAM* directement attachée au processeur *RM9000*. Malheureusement, cette mémoire n'est pas accessible depuis le contrôleur *Ethernet*, qui ne peut «voir» que la mémoire qui lui est directement attachée, à savoir la *DRAM* de 128 Mega-octets.

4.7.2 Implémentation

Tout d'abord, la solution de modifier les fonctions d'allocation et de libération des *sockets buffers* a été envisagée. Toutefois, ce code est indépendant de la plateforme et de l'architecture, et nous n'avons donc pas trouvé souhaitable de le modifier. D'autre part, nous ne connaissions pas les répercussions exactes que pourraient avoir les modifications opérées sur ces fonctions dans le fonctionnement de la pile de protocoles réseau.

La solution qui a été retenue consiste à allouer séparément les *buffers DMA* des *sockets buffers*. Cette solution nécessite une copie mémoire supplémentaire : à l'émission, d'un *socket buffer* vers un *DMA buffer* et à la réception, d'un *DMA buffer* vers un *socket buffer*. Bien que sous-optimale, cette solution est suffisante dans le cadre de cette plateforme : l'interface réseau du *Marvell* est utilisée uniquement à des fins de déboguage et de *monitoring*. Elle ne fait pas partie du chemin de données par lequel transite le traffic réseau traité par la plateforme, et on peut donc se contenter de performances sous-optimales.

Allocateur de buffers DMA

Les *buffers DMA* doivent être alloués dans la *DRAM* attachée au contrôleur *Marvell*. Puisque cette mémoire n'est pas gérée par le noyau Linux, il a été nécessaire d'écrire un petit allocateur de mémoire. Celui-ci est implémenté dans les fichiers `arch/mips/npp-board/dma-alloc.c` et `arch/mips/npp-board/mv64340-eth-fix.c`.

Le premier fichier contient l'implémentation de l'allocateur lui-même. Celui-ci gère une zone de mémoire démarrant à l'adresse physique `NPP_BOARD_MARVELL_DRAM_ETHERNET_RESERVED_PHYS_BASE` et d'une taille de `NPP_BOARD_MARVELL_DRAM_ETHERNET_RESERVED_SIZE` octets (voir `include/asm-mips/npp-board/board.h` pour les définitions). Cette zone est découpée en deux régions, gérées de manières différentes par l'allocateur.

Allocation : première région La première région est utilisée pour allouer l'espace nécessaire pour les descripteurs de buffer DMA. Cette région occupe les `ETHERNET_DMA_DESC_AREA_SIZE` premiers octets de la zone réservée pour l'Ethernet (64 Ko par défaut). Les descripteurs DMA sont alloués une seule et unique fois à l'initialisation de l'interface réseau, en deux blocs : un bloc pour les descripteurs utilisés en émission, un autre bloc utilisé en réception. La stratégie d'allocation choisie pour cette région est très simple : chaque allocation a pour effet d'incrémenter un pointeur, et la libération n'a aucun effet. Il n'est donc pas possible de démarrer et stopper l'interface réseau de nombreuses fois avec la commande `ifconfig`. Néanmoins, dans le cadre d'utilisation de la plateforme, cela n'est pas un problème majeur.

La variable `region_alloc_addr` est initialement positionnée à l'adresse du début de la région réservée pour l'allocation des descripteurs de buffers DMA, et `region_alloc_remaining_sz` à la taille de cette région. Au fil des allocations avec la fonction `dma_alloc_region`, `region_alloc_addr` est incrémenté et `region_alloc_remaining_sz` est décrémenté. La fonction `dma_free_region`, bien que présente ne fait rien.

Allocation : deuxième région La deuxième région, la plus importante en taille, couvre la deuxième partie de la zone réservée à l'Ethernet dans la DRAM attachée au contrôleur Marvell. Cette région est découpée en blocs de taille fixe, suffisante pour accueillir n'importe quel paquet réseau. La taille d'un bloc est de `NPP_BOARD_ETHERNET_DMA_BUFFER_SIZE` octets, valeur définie dans `include/asm-mips/mv64340/mv64340_eth.h`.

Chaque bloc libre contient une structure de type `free_element`, qui permet de chaîner les blocs libres entre eux.

```
struct free_element {
#ifdef DMA_MALLOC_DEBUG
  unsigned magic1;
#endif
  struct free_element *next;
#ifdef DMA_MALLOC_DEBUG
  unsigned magic2;
#endif
};
```

À l'initialisation (fonction `dma_alloc_init`), une liste chaînée de tous les blocs est construite. La liste chaînée est accessible via la variable globale `buffer_list`. La fonction `dma_alloc_buffer` permet d'allouer un buffer DMA en le prenant en tête de la liste, et la fonction `dma_free_buffer` permet de libérer un buffer en le réinsérant en tête de liste.

La variable de compilation `DMA_MALLOC_DEBUG` permet d'ajouter avant et après le pointeur de chaînage `next` dans la structure `free_element` des valeurs «magiques». Celles-ci permettent de vérifier qu'il n'y a pas eu écrasement d'informations.

Calcul d'adresses et copie

Le pilote Ethernet ne fait pas directement appel aux fonctions d'allocation et de libération de buffer DMA. Il passe par un autre jeu de fonctions s'occupant à la fois de l'allocation des buffers DMA mais également de la recopie des données de ou vers les *socket buffers*. Ces fonctions sont implémentées dans le fichier `arch/mips/npp-board/mv64340-eth-fix.c`.

Le premier jeu de fonctions sert à traduire les adresses des buffers et des descripteurs. En effet, le processeur travaille avec des adresses virtuelles dans les segments KSEG0 ou KSEG1, alors que le contrôleur Ethernet fonctionne avec des adresses physiques relatives au début de la DRAM du *Marvell*. D'autre part, les *buffers DMA* sont accédés au travers de `KSEG0`, afin de bénéficier du cache. Les *descripteurs* en revanche, sont accédés au travers de `KSEG1`, pour éviter à avoir à manipuler le cache.

Le deuxième jeu de fonctions permet de copier les données depuis les *socket buffer* vers les *buffer DMA* ou l'inverse, et d'allouer et libérer les *buffers DMA*. Ces fonctions se chargent aussi d'invalider le cache ou de synchroniser le cache du processeur avec la mémoire lorsque cela est nécessaire. En effet, le contrôleur Ethernet accède directement à la *DRAM*, sans passer par le cache processeur. Il faut donc s'assurer de la cohérence des accès, c'est à dire qu'il faut :

- Après une copie d'un *socket buffer* dans un *buffer DMA* effectuer une opération *write-back* sur le cache pour s'assurer que le contenu de celui-ci est bien écrit dans la mémoire avant de lancer le transfert réseau ;
- Avant une copie d'un *DMA buffer* dans un *socket buffer* effectuer une opération *invalidate* sur le cache pour s'assurer que le prochain accès à la zone se fera directement en mémoire en ignorant les éventuelles lignes de cache.

4.7.3 Modification du pilote réseau

La modification du pilote réseau a principalement consisté à :
- Initialiser correctement l'interface réseau en récupérant l'adresse MAC directement depuis la carte plutôt que depuis la mémoire *Flash* ;
- Ajouter avant une émission et avant une réception, les appels aux fonctions d'allocation/libération et de copie des *buffers DMA*

La réalisation de ces modifications a nécessité la compréhension complète du fonctionnement d'un pilote d'interface réseau, dont l'implémentation totalise environ 3000 lignes de code. Il était en effet nécessaire de s'assurer qu'à tous les endroits du pilote les modifications appropriées étaient effectuées, ce qui a nécessité de longues étapes de déboguage.

4.7.4 Améliorations possibles

Les modifications du pilote réseau sont améliorables sur différents points :
- L'option de *checksum offloading*, qui permet de déporter le calcul des *checksums* des paquets IP et TCP au matériel, a été désactivée car elle semblait poser des problèmes. Il faudrait étudier plus en détails l'origine de ces problèmes et éventuellement les corriger ;
- L'option de *new network API* a été désactivée. La nouvelle *API* d'un pilote réseau, apparue avec le noyau 2.6, permet de désactiver les interruptions pour fonctionner en mode *polling* lorsque le nombre d'interruptions devient très important, lors d'un afflux massif de paquets. Il faudrait appliquer les modifications appropriées aux fonctions implantant cette nouvelle API dans le pilote.

Chapitre 5

Fonctionnement multi-processeur

5.1 Approches possibles

Le processeur *MIPS* est un processeur à double cœur, de type *Symmetric Multiprocessing* (SMP), c'est à dire que l'accès à la mémoire est symétrique : les deux cœurs ont accès à la même mémoire. Jusqu'ici, un seul cœur était utilisé pour l'exécution du noyau Linux. L'équipe souhaitait pouvoir bénéficier du second cœur pour effectuer plus de traitements.

J'ai tout d'abord commencé par réaliser une petite étude des différents moyens existants pour bénéficier du second processeur et répondre aux demandes de l'équipe :

- Utilisation du noyau Linux standard en multi-processeur, afin de répartir le travail des processus sur les deux cœurs ;
- Utilisation d'un noyau Linux au-dessus d'un nano-noyau temps réel, comme ce que propose le projet Adeos[1] ;
- Utilisation du deuxième processeur sans système d'exploitation pour exécuter directement du code.

Nous avons également effectués quelques calculs en ce qui concerne les besoins en réactivité de la partie *MIPS* de la plateforme. La partie *ARM* peut transmettre en pointe des paquets de 64 octets à la vitesse de 1 Gbit/s, soit jusqu'à un paquet toutes les 500 nano-secondes environ. Même un noyau temps-réel ne permet pas d'atteindre la réactivité nécessaire.

Après discussion avec l'équipe, c'est la troisième solution qui a été retenue. Elle permet de faire fonctionner sur le premier cœur le noyau Linux, qui effectue les traitements de haut niveau sur les paquets, et sur le deuxième cœur un code spécialisé qui peut récupérer les paquets et effectuer un traitement de bas niveau sur ceux-ci. En faisant fonctionner ce code directement sur le deuxième cœur, il n'y aucune surcharge dûe au fonctionnement d'un système d'exploitation, et il est possible d'optimiser au maximum le fonctionnement de ce code, par exemple en le bloquant dans le cache du processeur.

5.2 Implémentation

5.2.1 Démarrage

Lorsqu'un cœur démarre, il commence à exécuter le code présent à l'adresse `0xBFC00000`, c'est à dire au milieu de la mémoire *Flash*. Cette adresse est identique pour le premier et pour

[1] `http://home.gna.org/adeos/`

le second cœur.

A cette adresse `0xBFC00000` est située le premier niveau du chargeur de système d'exploitation, *PMON*. La version de *PMON* utilisée par l'équipe, et largement modifiée pour la plateforme, ne supportait pas le SMP. Si un deuxième cœur démarrait, il exécuterait le même code que le premier cœur, ce qui amènerait fatalement à des problèmes.

Il a donc été nécessaire de modifier le code d'initialisation du chargeur de système d'exploitation. Ce chargeur est un logiciel assez ancien, peu maintenu, dans lequel l'organisation des sources et du processus de compilation est difficilement compréhensible. Il a donc tout d'abord été nécessaire de comprendre le fonctionnement interne de ce logiciel. Le code d'initialisation étant écrit en assembleur MIPS, j'ai du apprendre rapidement les instructions principales et l'utilisation des registres sur cette architecture. Les modifications pour le support du deuxième cœur portent sur un seul fichier, `start.S`.

D'autre part, le code à l'adresse `0xBFC00000` est situé dans la mémoire *Flash*, à chaque essai, il est donc nécessaire de *flasher* cette mémoire. Il est donc préférable de limiter le nombre d'essais. D'autre part, toute erreur dans le code d'initialisation empêche le premier cœur de démarrer, et donc empêche de *reflasher* un chargeur fonctionnel. Dans ce cas il est nécessaire d'utiliser une sonde *JTAG*[2] pour prendre le contrôle matériellement du processeur et envoyer un code fonctionnel dans la mémoire. Malheureusement, le logiciel *Corelis* associé à la sonde étant d'assez mauvaise qualité, ce processus est parfois assez délicat à mettre en oeuvre, et de nombreuses tentatives sont nécessaires. Toute modification entreprise au niveau du *PMON* est donc une opération délicate, et en cas d'erreur il était parfois nécessaire de passer une demi-journée voire une journée pour arriver à reprendre le contrôle de la machine en utilisant la sonde *JTAG*.

D'autre part, les possibilités de déboguage à ce niveau sont quasi nulles : le logiciel associé à la sonde *JTAG* fonctionne de manière complètement aléatoire dès lors que les deux cœurs sont utilisés.

Le développement des quelques lignes d'assembleur nécessaires à l'initialisation du second cœur a donc été particulièrement délicat.

Branchement dans le code en fonction du cœur

Dans ce fichier, l'exécution du code commence au label `start`. Le début de ce code est commun aux deux cœurs et met simplement en place une pile et initialise divers registres. Peu après, l'ajout suivant, après le label `locate`, a été réalisé :

Listing 5.1 – Branchement entre le code du premier et du second cœur

```
#ifndef MIDROM
        /* Who are we? If not CPU 0 skip init */
        mfc0    v0, COP_0_PRID
        srl     v0, 24
        andi    v0, 0x07
        bnez    v0, start_other_cpu
        nop
#endif
```

Ce code teste sur quel cœur on se situe, et s'il s'agit du deuxième, saute au label `start_other_cpu`. A partir de là, les chemins d'exécution des deux cœurs diffèrent : le premier

[2] Une sonde *JTAG* est un appareil permettant de contrôler matériellement l'exécution du processeur.

poursuit l'exécution du code normalement, le second va exécuter du code spécifique.

Ajout de la *Scratch RAM*

Le processeur *RM9000* contient une petite mémoire, la *Scratch RAM*, d'une taille de 8 Ko et partagée entre les deux cœurs. Jusqu'ici, elle était non utilisée, j'ai donc rajouté le code nécessaire pour la mapper en mémoire, à l'adresse physique `0x1A000000`, ce qui la rend disponible via KSEG0 à l'adresse `0x9A000000` et via KSEG1 à l'adresse `BA000000` (voir figure 4.1). Ce code est exécuté par le premier cœur.

Listing 5.2 – Initialisation de la *Scratch RAM*

```
1           la      v1, RM9K_BASE_ADDR
2           li      v0, ((PKT_SRAM_SIZE-1) >> 4) & ~0xff
3           sw      v0, LKM13(v1)
4           li      v0, (PKT_SRAM_BASE >> 4) | 1
5           sw      v0, LKB13(v1)
6
7           la      v0, PHYS_TO_UNCACHED(PKT_SRAM_BASE)
8           addu    v1, v0, PKT_SRAM_SIZE
9   1:
10          addiu   v0, 8
11          bne     v0, v1, 1b
12          sd      zero, -8(v0)
```

Les lignes 1 à 5 permettent de mapper à l'adresse `PKT_SRAM_BASE` la *Scratch RAM* sur `PKT_SRAM_SIZE` octets (en réalité il s'agit de 8 Kilo-octets). Les lignes 7 à 12 permettent d'initialiser à 0 cette mémoire.

Initialisation du second cœur

L'initialisation proprement dite du second cœur commence au label `start_other_cpu`. Cette initialisation se déroule en plusieurs étapes (voir listing 5.3) :

- **Ligne 2** : initialisation d'une pile ;
- **Lignes 4 à 10** : initialisation des *TLB*, *Translation Lookaside Buffers* du second cœur. Les *TLB*s sont des caches de traductions d'adresses virtuelles → physiques ;
- **Lignes 12 et 13** : initialisation des *caches* (de niveau 1 et de niveau 2). À l'origine, elle était intégrée au code d'initialisation du premier cœur. Ce code a été factorisé au sein de la procédure `init_caches` de manière à être utilisable par les deux cœurs ;
- **Lignes 15 et 16** : calcul de la taille du code d'attente (*holding code*), stockée dans `v1`, puis saut au label `after_holding_code`. En réalité, la ligne `16` est exécutée «avant» la ligne `15`, en raison d'une spécificité des processeurs MIPS, le *delay slot* ;
- L'exécution se poursuit **ligne 36**. Les **lignes 37 à 44** servent à recopier le code d'attente (*holding code*) depuis là où il est stocké vers la *Scratch RAM*. L'adresse du *holding code* est déterminée par le registre `ra`, initialisé lors du précédent saut réalisé par l'instruction `bal`. L'adresse de destination est l'adresse de début de la *Scratch RAM* incrémentée de 4. Le premier mot de la *Scratch RAM* est réservé pour une autre utilisation ;
- **Lignes 46 et 47**, le premier mot de la *Scratch RAM* est initialisé à 0 ;
- **Lignes 49 et 50**, on saute dans le code d'attente stocké au début de la *Scratch RAM* ;

- L'exécution se poursuit **ligne 18**, mais le code n'est plus exécuté depuis la *Flash*, mais depuis la *Scratch RAM*. Tout d'abord, **lignes 19 à 21**, on positionne à 1 la 31ème sémaphore du registre de sémaphores matérielles. Ce registre est commun aux deux cœurs, ce qui leur permet de se synchroniser ;
- **Lignes 23 à 27**, on boucle tant que la 31ème sémaphore est à 1, c'est à dire tant que le premier cœur n'a pas signalé qu'il voulait que l'exécution se poursuive dans le deuxième cœur ;
- **Lignes 29 à 32**, le deuxième cœur a été débloqué, il charge le contenu du premier mot de la *Scratch RAM* et saute à cette adresse. Le premier cœur doit donc avoir initialisé ce mot pour qu'il contienne l'adresse du code à exécuter par le deuxième cœur et ce, avant de relâcher la sémaphore.

Listing 5.3 – Code d'initialisation du second cœur

```
1  start_other_cpu:
2          la      sp, stack-16384
3
4          bal     CPU_TLBClear
5          nop
6
7          li      a0, 0xc0000000
8          li      a1, 0x40000000
9          bal     CPU_TLBInit
10         nop
11
12         bal     init_caches
13         nop
14
15         bal     after_holding_code
16         ori     v1, zero, holdingcodesz-holdingcode
17
18 holdingcode:
19         la      s2, RM9K_BASE_ADDR
20         li      v0, 0x80000000
21         sw      v0, RM9000_SEMSET(s2)
22
23 holdwait:
24         lw      v1, RM9000_SEM(s2)
25         and     v1, v0
26         bnez    v1, holdwait
27         nop
28
29         la      v0, (PHYS_TO_UNCACHED(PKT_SRAM_BASE))
30         lw      v1, (v0)
31         jr      v1
32         nop
33
34 holdingcodesz:
35
```

```
after_holding_code:
        la      v0, (PHYS_TO_UNCACHED(PKT_SRAM_BASE)+4)
2:
        lw      t0, (ra)
        sw      t0, (v0)
        addiu   v1, -4
        addiu   v0, 4
        bgtz    v1, 2b
        addiu   ra, 4

        la      v0, (PHYS_TO_UNCACHED(PKT_SRAM_BASE))
        sw      zero, (v0)

        la      v0, (PHYS_TO_UNCACHED(PKT_SRAM_BASE)+4)
        jr      v0
        nop
```

Ce code permet donc au second cœur de s'initialiser puis de sauter à une adresse définie par le premier cœur. Du point de vue du premier cœur, le processus d'initialisation du second est le suivant :

1. Sortir le second cœur de l'état *reset*, de manière à ce que celui-ci commence l'exécution de son code d'initialisation ;
2. Attendre que la 31ème sémaphore passe à 1, ce qui permet de vérifier que le second cœur a correctement démarré ;
3. Écrire au début de la *Scratch RAM* l'adresse du code à exécuter par le second cœur ;
4. Passer la 31ème sémaphore à 0, ce qui va relâcher le second cœur qui poursuivra son exécution à l'adresse précisée précédemment.

5.2.2 Support du *SMP*

Le code d'initialisation étudié dans la partie précédente permet donc au premier cœur de démarrer le second et de lui faire exécuter un code personnalisé. Toutefois, il faut automatiser ce processus de démarrage et faciliter le chargement du code personnalisé en mémoire.

Architecture générale

La première solution envisagée a été de créer un module noyau qui contiendrait à la fois le code démarrant le second cœur et le code exécuté par ce second cœur. Cette solution a été expérimentée, mais il s'est rapidement avéré qu'elle ne convenait pas : sur MIPS, les modules noyau sont chargés dans la partie `kseg2` de l'espace d'adressage. Or, cette partie de l'espace d'adressage passe par la MMU, et donc par les *TLB*, ce qui nécessite l'intervention du système d'exploitation. Cela fonctionne sur le premier cœur car Linux s'en occupe, mais sur le second cœur il n'y a rien pour configurer les TLB.

Finalement, la solution adoptée a été d'avoir une architecture en trois éléments distincts :

– Un **module noyau** qui permet de démarrer et d'arrêter le second cœur et propose un périphérique caractère spécifique, `cpu-loader` qui permet à une application utilisateur d'interagir avec le noyau pour la gestion du second cœur ;

– Une **application utilisateur** qui permet via le périphérique caractère `cpu-loader` de charger du code pour le second cœur, de le lancer et de l'arrêter ;
– Le **code** exécute par le second cœur lui-même.

Pour que ceci fonctionne, une zone de la mémoire a été réservée pour le fonctionnement du deuxième cœur. Cette zone se situe dans la DRAM attachée au processeur. Le noyau Linux ne gère donc pas en réalité les 256 Mo de cette mémoire, mais un peu moins.

Dans la configuration actuelle, 255 Mo des 256 Mo sont réservés pour le noyau Linux et le Mega-octet restant est consacré au fonctionnement du deuxième cœur. Le fichier `include/asm-mips/mach-npp/board.h` définit diverses constantes telles que `NPP_BOARD_RM9000_DRAM_LINUX_SIZE` et `NPP_BOARD_RM9000_DRAM_SECOND_CORE_SIZE` qui délimitent ces zones.

Ces éléments sont implémentés en dehors du noyau, dans un ensemble appelé `npp-smp-support`, découpé en trois répertoires, `kernel` pour le module noyau (voir partie 5.2.2), `user` pour la partie utilisateur (voir partie 5.2.2) et `code` pour le code lui-même (voir partie 5.2.2).

Module noyau

Le module noyau est constitué de deux fichiers sources, `kernel/cpu-loader-main.c` et `kernel/cpu-support.c`. Le second est utilisé pour l'affichage de messages entre les deux cœurs, fonctionnalité qui sera étudiée dans la partie 5.2.3. Ces deux fichiers sont compilés au sein d'un unique fichier objet `cpu-loader.ko` qui constitue le module noyau, et que l'on charge sur la plateforme en utilisant la commande `insmod`.

Le premier fichier, `kernel/cpu-loader-main.c`, implémente le périphérique caractère `cpu-loader`. La structure `cpu_loader_fops` liste les opérations supportées par ce périphérique caractère, à savoir `write`, `ioctl`, `open` et `release`.

Initialisation La fonction `init_module`, appelée à l'initialisation du module, s'occupe simplement d'enregistrer le périphérique caractère et d'initialiser des données.

Ouverture La fonction `cpu_loader_open` est chargée de vérifier que le périphérique caractère n'est pas déjà ouvert, puis d'initialiser quelques variables de la structure `cpu_loader_infos`. En particulier, elle initialise le champ `ptr` de cette structure à `CODE_LOAD_ADDR`, l'adresse à laquelle débutera le chargement du code.

Fermeture La fonction `cpu_loader_close` marque simplement que le périphérique est maintenant fermé.

Écriture La fonction `cpu_loader_write` est appelée lorsque des données sont écrites sur le périphérique caractère. Elle permet de charger le code en mémoire, dans la zone réservée à cet effet.

Démarrage et arrêt La fonction `cpu_loader_ioctl` est l'implémentation de l'appel système classique sous Unix `ioctl`. Il permet d'implémenter des fonctionnalités spécifiques au périphérique. Dans notre cas, trois fonctionnalités sont proposées : `IOCTL_CORE_START` qui permet de démarrer le deuxième cœur, `IOCTL_CORE_RESET` qui permet d'arrêter le

deuxième cœur et `IOCTL_CORE_STATUS` qui permet d'obtenir des informations sur l'état du deuxième cœur, par l'intermédiaire d'une structure `smp_status`. Ces trois fonctionnalités sont respectivement implémentées dans les fonctions `cpu_loader_start_code`, `cpu_loader_stop_code` et `cpu_loader_get_status`. La fonction `cpu_loader_start_code` est celle qui s'occupe de démarrer le deuxième cœur en respectant le protocole décrit dans les sections précédentes, à savoir : **1)**, sortie de l'état *reset*, **2)** attente de l'activation à 1 de la 31ème sémaphore, **3)**, écriture de l'adresse de début du code et **4)** relâchement de la 31ème sémaphore pour déclencher l'exécution du code.

Application utilisateur

L'application utilisateur qui permet de charger le code, de lancer et d'arrêter le deuxième cœur est implémentée dans le fichier `user/cpu-loader-user.c`. Elle utilise `getopt` pour lire les options de la ligne de commande, puis ouvre le périphérique caractère `/dev/cpu-loader` avant de réaliser les opérations demandées par l'utilisateur, qui peuvent être :

- **Chargement du fichier**, option `-l` suivie du nom du fichier à charger. Tout d'abord, l'application vérifie que le second cœur n'est pas en fonctionnement, puis appelle `cpu_loader_load_file` qui se charge d'ouvrir le fichier, et de réaliser autant d'appels `write()` que nécessaire pour transmettre le code au module noyau ;
- **Démarrage du cœur**, option `-s`. L'application vérifie que le cœur n'est pas déjà en fonctionnement et que du code à exécuter est chargé. Si oui, alors le deuxième cœur est démarré en utilisant `cpu_loader_start_code` qui elle-même repose sur l'appel `ioctl IOCTL_START_CORE` ;
- **Arrêt du cœur**, option `-r`. L'application vérifie que le cœur est en exécution, et si c'est le cas, elle appelle la fonction `cpu_loader_reset_core` pour l'arrêter. Cette dernière utilise l'appel `ioctl IOCTL_RESET_CORE` ;
- **Informations sur le cœur**, option `-i`. L'application réalise l'appel `ioctl IOCTL_CORE_STATUS` qui remplit une structure de type `smp_status`. La définition de cette structure est dans un fichier d'entête commun à l'application et au module noyau, `cpu-loader-common.h`.

La suite de commandes suivante présente une utilisation typique du module noyau et de l'application utilisateur, en supposant que le fichier contenant le code du second coeur s'appelle `code-coeur` :

```
# insmod cpu-loader.ko
# ./cpu-loader-user -i
Processor is NOT running, code NOT loaded
# ./cpu-loader-user -l code-coeur
# ./cpu-loader-user -i
Processor is NOT running, code loaded
# ./cpu-loader-user -s
# ./cpu-loader-user -i
Processor is running, code loaded
# ./cpu-loader-user -r
# ./cpu-loader-user -i
Processor is NOT running, code loaded
```

Ainsi, il est possible sans redémarrer la plateforme, de charger du code différent, et de démarrer et arrêter plusieurs fois le deuxième cœur.

Code à exécuter

Le code s'exécutant sur le deuxième cœur s'exécute sans aucun système d'exploitation. Il s'agit de code C ainsi que d'un peu de code assembleur d'initialisation, relogé à l'aide d'un éditeur de liens pour fonctionner à l'adresse prévue.

Initialisation Le fichier `start.S` contient le code d'initialisation du second cœur, avant que le code C ne soit appelé. Il est stocké dans une section spécifique du binaire appelée `bootstrap`:

Listing 5.4 – Code d'initialisation du code du second cœur

```
1         .set    noreorder
2         .section "bootstrap"
3         .globl  _start
4
5         li $29, 0xAFFFF000
6         nop
7
8         mfc0    $27, $12
9         li      $26, 0xFFBFFFFF
10        and     $27, $27, $26
11        mtc0    $27, $12
12
13        j code_main
14        nop
15  loop:
16        b loop
```

Les lignes 5 et 6 initialisent une pile située en haut de la zone réservée pour le deuxième cœur dans la mémoire physique. Les lignes 8 à 11 changent le mode de fonctionnement des gestionnaires d'interruption. En effet, à l'initialisation du deuxième cœur, les gestionnaires d'interruptions sont par défaut situés dans la *Flash*. Ici, on modifie la configuration du processeur pour que ces gestionnaires soient situés dans la *mémoire vive*. Enfin, ligne 13, on appelle la fonction principale du code C, `code_main`.

Code C Le code C lui-même ne présente rien de particulier, il doit effectuer les opérations que l'on souhaite faire effectuer au second cœur. Il doit simplement définir une fonction `void code_main(void)`. Toutefois, quelques fonctionnalités ont été ajoutées pour faciliter le déboguage, voir partie 5.2.3.

Compilation Un *script de linker* est utilisé pour indiquer à l'éditeur de liens comment reloger le code pour qu'il s'exécute correctement à l'adresse où il est chargé. Ce *script* est stocké dans le fichier `code.lds`. Il décrit l'organisation des sections du binaire et leur localisation en mémoire.

Les lignes 1 et 2 précisent que le code de sortie doit être au format *big endian*, *ELF* pour architecture *MIPS*. La ligne 5 précise que l'adresse courante de chargement est `0xAFF00000`, ce qui correspond au début de la zone réservée au second cœur dans la mémoire physique. Les lignes 7 à 14 précisent l'organisation de la section contenant le code. Celle-ci doit d'abord

regrouper le code de la section `exception`, qui sera étudiée en partie 5.2.3, puis la section `bootstrap`, puis le code C (section `.text`) puis les données en lecture seule (`.rodata`). On note que la section `bootstrap` est alignée sur `0x100` octets, donc en réalité elle débute à l'adresse `0xAFF00100` (car la section `exception` ne doit pas faire plus de `0x100` octets). Le reste du fichier décrit l'organisation des autres sections.

Listing 5.5 – Code d'initialisation du code du second cœur

```
1  OUTPUT_FORMAT(elf32-tradbigmips)
2  OUTPUT_ARCH(mips)
3  SECTIONS
4  {
5    . = 0xaff00000;
6
7    _text = .;
8    .text : {
9      *(exception)
10     . = ALIGN(0x100);
11     *(bootstrap)
12     *(.text)
13     *(.rodata)
14   }
15
16   _etext = .;
17
18   [...]
19
20   _end = . ;
21 }
```

Suite à sa compilation au format *ELF* grâce à l'éditeur de liens et au *script de linkage*, le binaire est converti au format *binaire brut*. En effet, le chargeur de code ne supporte pas le format *ELF*, uniquement le binaire brut. Pour réaliser cette conversion, la commande `objcopy` est utilisée dans le `Makefile`.

5.2.3 Déboguage

Depuis le premier cœur, on ne dispose quasiment d'aucun moyen de contrôle du second cœur en dehors du démarrage et de l'arrêt. Pourtant, afin de faciliter la mise au point du code exécuté sur le deuxième cœur, il est indispensable d'avoir des fonctionnalités de déboguage : l'*affichage de messages* et la *gestion des erreurs* par les interruptions.

Affichage de messages

L'affichage de message par le deuxième cœur s'effectue en passant par le premier cœur : une zone de la *Scratch RAM* est réservée pour stocker une chaîne de caractère que le second cœur veut afficher (voir figure 5.1). Cette zone est couverte par la structure `msg_exchange_area` définie dans le fichier `cpu-loader-common.h` :

```
#define MSG_EXCHANGE_AREA_OWNER_CORE0 0
```

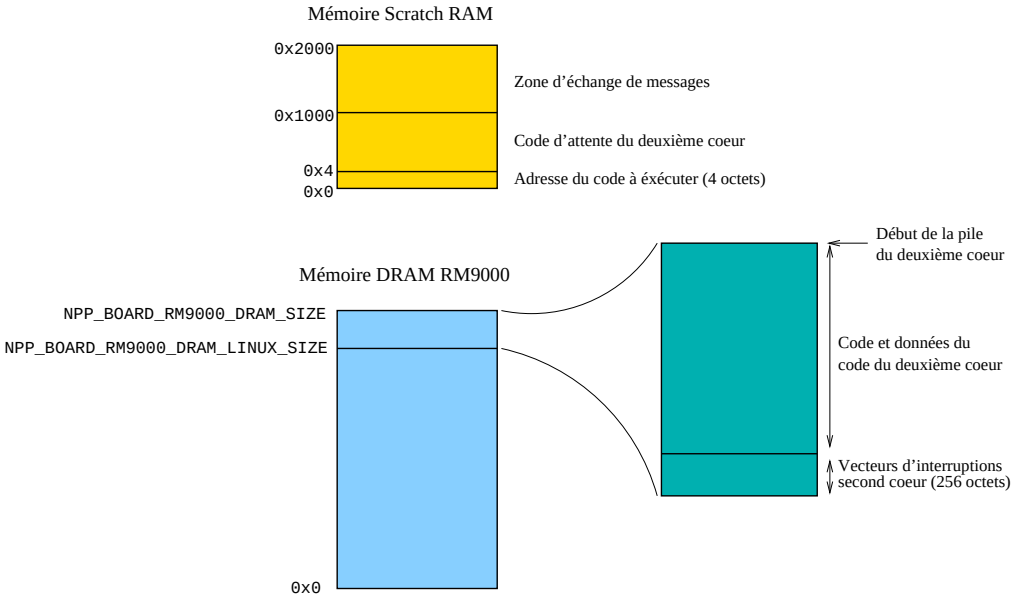

FIG. 5.1 – Utilisation de la mémoire pour le fonctionnement multi-processeur de la plate-forme

```
#define MSG_EXCHANGE_AREA_OWNER_CORE1 1

struct msg_exchange_area
{
  int owner;
  char msg[MSG_EXCHANGE_AREA_MAX_LEN];
}
```

Ainsi, lorsque `owner` vaut `MSG_EXCHANGE_AREA_OWNER_CORE1`, la zone «appartient» au cœur 1 et peut être utilisée pour y copier la chaîne de caractère à afficher. Une fois la copie effectuée, le cœur positionne `owner` à `MSG_EXCHANGE_AREA_OWNER_CORE0`. Le premier cœur sur lequel le noyau Linux fonctionne surveille de temps à autre cette variable, et lorsqu'il s'aperçoit qu'elle vaut `MSG_EXCHANGE_AREA_OWNER_CORE0`, il affiche la chaîne de caractère associée, puis repositionne `owner` à `MSG_EXCHANGE_AREA_OWNER_CORE1`.

Au niveau noyau Linux, le code implémentant cette fonctionnalité se trouve dans le fichier `kernel/cpu-support.c`. Un *timer* est configuré pour appeler la fonction `mea_timer` toutes les 40 milli-secondes. Les fonctions `mea_start` et `mea_stop` permettent respectivement de démarrer et d'arrêter le *timer* lorsque l'on démarre ou arrête le deuxième cœur.

Au niveau du second cœur, le code implémentant cette fonctionnalité se trouve dans le fichier `code/mea.c`. La fonction `mea_wait` permet de s'assurer que la zone d'échange est disponible en attendant éventuellement que le premier cœur la libère. La fonction `mea_commit` signale au premier cœur qu'il faut afficher la chaîne stockée dans la zone d'échange. La fonction `mea_putchar` ajoute un caractère à la fin de la chaînée stockée dans la zone d'échange.

À partir de ces fonctionnalités de base, il a été possible d'écrire une fonction `printk` simple dans le fichier `code/libcrt.c`. Ainsi, le code du deuxième cœur peut effectuer de manière très simple des affichages de chaînes de caractères ou de variables entières, les messages étant affichés sur la console série par le premier cœur.

Gestion des interruptions

La deuxième fonctionnalité importante pour permettre de développement de code sur le second cœur est la gestion des interruptions. En effet, étant donné qu'aucun système d'exploitation ne fonctionne sur ce cœur, les exceptions déclenchées en cas de violation de mémoire, de division par zéro ne sont pas gérées.

Sur MIPS, l'ensemble des exceptions est traité par un seul vecteur d'interruption. Il peut être soit localisé dans la *Flash*, soit dans la *RAM*. Étant donné que les modifications du code de la *Flash* (le *PMON*) sont peu aisées, il a été choisi de les stocker dans la *RAM*. Cette configuration est l'objet d'une partie du code d'initialisation présent dans le fichier `code/start.S` décrit plus haut.

Toutefois, l'adresse du vecteur d'interruption en *RAM* est identique pour les deux cœurs. Il faut donc modifier la procédure exécutée en cas d'exception pour qu'elle réalise un traitement spécial si l'exception est dûe au deuxième cœur. Pour cela, il a fallu modifier le noyau Linux, qui se charge du traitement des exceptions du premier cœur, et donc a fortiori du deuxième puisque les vecteurs sont à la même adresse.

Le vecteur d'exception est présent dans le fichier assembleur `arch/mips/kernel/genex.S`. Plus précisement, il s'agit de la fonction `except_vec3_generic`. Au début de cette fonction a été ajouté du code qui teste quel est le processeur courant (ligne 2 à 4). Si le processeur courant est le premier cœur alors l'exécution se poursuit normalement (ligne 5). Sinon, il y a saut à l'adresse du début de la zone réservée en *RAM* pour le deuxième cœur (ligne 8 et 9).

Listing 5.6 – Modification du gestionnaire d'exception du noyau Linux

```
1  #ifdef CONFIG_NPP_BOARD
2          mfc0    k0, CP0_PRID
3          srl     k0, 24
4          andi    k0, 0x07
5          beqz    k0, vec3_cpu0
6          nop
7
8          li      k0, KSEG0ADDR(NPP_BOARD_RM9000_DRAM_SECOND_CORE_BASE)
9          j       k0
10
11 vec3_cpu1:
12         b       vec3_cpu1
13         nop
14 vec3_cpu0:
15 #endif
```

Au début de la zone réservée au second cœur se trouve le contenu de la section `exception` dont il a été question plus haut dans la description du *script d'édition de liens*. Cette section contient le code chargé de gérer les exceptions sur le second cœur, implémenté dans le fichier `code/start.S`:

Listing 5.7 – Gestionnaire d'exception du second cœur

```
1          .section "exception"
2
3          /* Read CP0_CAUSE ($13) in a0 ($4) */
```

```
4           mfc0    $4, $13
5           mfc0    $5, $14
6           mfc0    $6, $15
7           mfc0    $7, $12
8           la      $27, exception_handler
9           jr      $27
10
11  exloop:
12          b exloop
13          nop
```

Le code commence par lire le contenu de différents registres d'état qui contiennent la cause de l'exception, l'adresse de code à laquelle elle a eu lieu, l'adresse qui a déclenché l'exception (lors d'un accès mémoire par exemple), ainsi que l'état du processeur. Ces informations sont passées en paramètre à la fonction `exception_handler` présente dans le fichier `code/code.c`. Celle-ci, en fonction de l'origine de l'exception, affiche un message en utilisant la fonctionnalité de passage de message décrite en section 5.2.3.

5.3 Améliorations possibles

De nombreuses améliorations sont possibles concernant le support *SMP* sur la plateforme *NPP* :

- simplifier le processus d'initialisation du deuxième cœur au niveau du *PMON*. Il est probablement possible de s'affranchir de la recopie du code d'attente dans la *Scratch RAM* et de l'exécuter directement em *Flash* ;
- modifier le *script d'édition de liens* ainsi que tout code fixant l'adresse de la zone de mémoire réservée au second cœur ;
- améliorer le mécanisme d'échange de messages entre les deux cœurs en utilisant un système à plusieurs *buffers* ;
- mettre en place des mécanismes de communication de données plus évolués entre les deux cœurs, par exemple basés sur des files de buffers manipulées sans synchronisation.

Conclusion

Mon stage dans l'équipe *Protocoles* de *Mitsubishi Electric* m'a permis de travailler dans un domaine qui m'intéresse particulièrement : le développement système. En particulier, j'ai pu approfondir mes connaissances des outils de développement libres, ainsi que du noyau Linux. D'autre part, le contexte de la plateforme *FlexNP* a été l'occasion de découvrir des problématiques réseau et de travailler dans un environnement matériel particulier.

Par ailleurs, le contact avec les ingénieurs *hardware* réalisant le développement matériel de la plateforme a été particulièrement enrichissant. Ce stage a été l'occasion de découvrir le mode de travail d'ingénieurs d'un autre domaine, et d'échanger avec des personnes ayant une vision différente des projets et des développements.

Pour l'équipe, le travail réalisé durant le stage a permis d'aboutir à la mise en place d'outils de développement de meilleure qualité, au portage d'un noyau Linux récent, et surtout au développement d'une architecture pour l'utilisation du second cœur du processeur *MIPS*.

Enfin, sur le plan humain, le stage s'est particulièrement bien déroulé. L'intégration dans l'équipe a été très rapide, et les échanges au cours du stage très nombreux. Plusieurs étapes du développement et de réflexion sur des points de conception ont ainsi été menées en collaboration avec des autres membres de l'équipe.

Annexes

Liste des annexes :

Annexe A *Buildroot* documentation ;

Annexe B Interrupt handling with the Linux kernel for the NPP board.

Annexe A

Buildroot documentation

Cette documentation a été rédigée dans le cadre de mon stage, et a été intégrée au projet Buildroot.

Usage and documentation by Thomas Petazzoni. Contributions from Karsten Kruse, Ned Ludd and Martin Herren.

About Buildroot

Buildroot is a set of Makefiles and patches that allows to easily generate both a cross-compilation toolchain and a root filesystem for your target. The cross- compilation toolchain uses uClibc [1], a tiny C standard library.

Buildroot is useful mainly for people working with embedded systems. Embedded systems often use processors that are not the regular x86 processors everyone is used to have on his PC. It can be PowerPC processors, MIPS processors, ARM processors, etc.

A compilation toolchain is the set of tools that allows to compile code for your system. It consists of a compiler (in our case, *gcc*), binary utils like assembler and linker (in our case, *binutils*) and a C standard library (for example *GNU Libc*, *uClibc* or *dietlibc*). The system installed on your development station certainly already has a compilation toolchain that you can use to compile application that runs on your system. If you're using a PC, your compilation toolchain runs on an x86 processor and generates code for a x86 processor. Under most Linux systems, the compilation toolchain uses the GNU libc as C standard library. This compilation toolchain is called the "host compilation toolchain", and more generally, the machine on which it is running, and on which you're working is called the "host system". The compilation toolchain is provided by your distribution, and Buildroot has nothing to do with it.

As said above, the compilation toolchain that comes with your system runs and generates code for the processor of your host system. As your embedded system has a different processor, you need a cross-compilation toolchain : it's a compilation toolchain that runs on your host system but that generates code for your target system (and target processor). For example, if your host system uses x86 and your target system uses ARM, the regular compilation toolchain of your host runs on x86 and generates code for x86, while the cross-compilation toolchain runs on x86 and generates code for ARM. Even if your embedded system uses a x86 processor, you might interested in Buildroot, for two reasons :

[1] `http://www.uclibc.org/`

– The compilation toolchain of your host certainly uses the GNU Libc which is a complete but huge C standard library. Instead of using GNU Libc on your target system, you can use uClibc which is a tiny C standard library. If you want to use this C library, then you need a compilation toolchain to generate binaries linked with it. Buildroot can do it for you.
– Buildroot automates the building of a root filesystem with all needed tools like busybox. It makes it much easier than doing it by hand.

You might wonder why such a tool is needed when you can compile gcc, binutils, uClibc and all the tools by hand. Of course, doing so is possible. But dealing with all configure options, with all problems of every gcc or binutils version it very time-consuming and uninteresting. Buildroot automates this process through the use of Makefiles, and has a collection of patches for each gcc and binutils version to make them work on most architectures.

Obtaining Buildroot

Buildroot is available as daily CVS snapshots or directly using CVS. The latest snapshot is always available at `http://uclibc.org/downloads/snapshots/buildroot-snapshot.tar.bz2`, and previous snapshots are also available at `http://uclibc.org/downloads/snapshots/`xs.

To download Buildroot using CVS, you can simply follow the rules described on the "Accessing CVS"-page (`http://www.uclibc.org/cvs_anon.html`) of the uClibc website (`http://www.uclibc.org`), and download the buildroot CVS module. For the impatient, here's a quick recipe :

```
$ cvs -d:pserver:anonymous@uclibc.org:/var/cvs login
$ cvs -z3 -d:pserver:anonymous@uclibc.org:/var/cvs co buildroot
```

Using Buildroot

Buildroot has a nice configuration tool similar to the one you can find in the Linux Kernel[2] or in Busybox[3]. Note that you can run everything as a normal user. There is no need to be root to configure and use Buildroot. The first step is to run the configuration assistant :

```
$ make menuconfig
```

For each entry of the configuration tool, you can find associated help that describes the purpose of the entry.

Once everything is configured, the configuration tool has generated a .config file that contains the description of your configuration. It will be used by the Makefiles to do what's needed.

Let's go :

```
$ make
```

This command will download, configure and compile all the selected tools, and finally generate a target filesystem. The target filesystem will be named `root_fs_ARCH.EXT` where `ARCH` is your architecture and EXT depends on the type of target filesystem selected in the Target options section of the configuration tool.

[2] `http://www.kernel.org/`
[3] `http://www.busybox.org/`

Customizing the target filesystem

There are two ways to customize the resulting target filesystem :

- Customize the target filesystem directly, and rebuild the image. The target filesystem is available under `build_ARCH/root/` where `ARCH` is the chosen target architecture. You can simply make your changes here, and run make afterwards, which will rebuild the target filesystem image. This method allows to do everything on the target filesystem, but if you decide to completely rebuild your toolchain and tools, these changes will be lost.
- Customize the target filesystem skeleton, available under `target/default/target_skeleton/`. You can customize configuration files or other stuff here. However, the full file hierarchy is not yet present, because it's created during the compilation process. So you can't do everything on this target filesystem skeleton, but changes to it remains even you completely rebuild the cross-compilation toolchain and the tools. You can also customize the `target/default/device_table.txt` file which is used by the tools that generate the target filesystem image to properly set permissions and create device nodes. The `target/default/skel.tar.gz` file contains the main directories of a root filesystem and there is no obvious reason for which it should be changed. These main directories are in an tarball instead of inside the skeleton because it contains symlinks that would be broken otherwise.

Customizing the *Busybox* configuration

Busybox is very configurable, and you may want to customize it. You can follow these simple steps to do it. It's not an optimal way, but it's simple and it works.

1. Make a first compilation of buildroot with busybox without trying to customize it.
2. Go into `build_ARCH/busybox/` and run make menuconfig. The nice configuration tool appears and you can customize everything.
3. Copy the `.config` file to `package/busybox/busybox.config` so that your customized configuration will remains even if you remove the cross- compilation toolchain.
4. Run the compilation of buildroot again.

Otherwise, you can simply change the package/busybox/busybox.config file if you know the options you want to change without using the configuration tool.

Customizing the uClibc configuration

Just like *BusyBox*, *uClibc* offers a lot of configuration options. They allow to select various functionalities, depending on your needs and limitations. The easiest way to modify the configuration of uClibc is to follow these steps :

1. Make a first compilation of buildroot without trying to customize uClibc.
2. Go into the directory `toolchain_build_ARCH/uClibc/` and run make menuconfig. The nice configuration assistant, similar to the one used in the Linux Kernel or in Buildroot appears. Make your configuration as appropriate.

3. Copy the `.config` file to `toolchain/uClibc/uClibc.config` or `toolchain/uClibc/uClibc.config-locale`. The former is used if you haven't selected locale support in Buildroot configuration, and the latter is used if you have selected locale support.
4. Run the compilation of Buildroot again.

Otherwise, you can simply change `toolchain/uClibc/uClibc.config` or `toolchain/uClibc/uClibc.config-locale` without running the configuration assistant.

How Buildroot works

As said above, Buildroot is basically a set of *Makefiles* that download, configure and compiles software with the correct options. It also includes some patches for various software, mainly the ones involved in the cross-compilation tool chain (*gcc*, *binutils* and *uClibc*).

There is basically one *Makefile* per software, and they are named with the `.mk` extension. *Makefiles* are split into three sections :

- *package* (in the `package/` directory) contains the Makefiles and associated files for all user-space tools that Buildroot can compile and add to the target root filesystem. There is one sub-directory per tool.
- *toolchain* (in the `toolchain/` directory) contains the Makefiles and associated files for all software related to the cross-compilation toolchain : *binutils*, *ccache*, *gcc*, *gdb*, *kernel-headers* and *uClibc*xs.
- *target* (in the `target/` directory) contains the Makefiles and associated files for software related to the generation of the target root filesystem image. Four types of filesystems are supported : *ext2*, *jffs2*, *cramfs* and *squashfs*. For each of them, there's a sub-directory with the required files. There is also a `default/` directory that contains the target filesystem skeleton.

Each directory contains at least 3 files :

- `something.mk` is the Makefile that downloads, configures, compiles and installs the software something.
- `Config.in` is a part of the configuration tool description file. It describes the option related to the current software.
- `Makefile.in` is a part of Makefile that sets various variables according to the configuration given through the configuration tool. For most tools it simply involves adding the name of the tool to the `TARGETS` variable.

The main Makefile do the job through the following steps (once the configuration is done) :

- Create the download directory (`dl/` by default). This is where the tarballs will be downloaded. It is interesting to know that the tarballs are in this directory because it may be useful to save them somewhere to avoid further downloads.
- Create the build directory (`build_ARCH/` by default, where `ARCH` is your architecture). This is where all user-space tools while be compiled.
- Create the toolchain build directory (`toolchain_build_ARCH/` by default, where ARCH is your architecture). This is where the cross compilation toolchain will be compiled.
- Setup the staging directory (`build_ARCH/staging_dir/` by default). This is where the cross-compilation toolchain will be installed. If you want to use the same cross-

compilation toolchain for other purposes, such as compiling third-party applications, you can add `build_ARCH/staging_dir/` bin to your `PATH`, and then use `arch-linux-gcc` to compile your application. In order to setup this staging directory, it first removes it, and then it creates various subdirectories and symlinks inside it.
- Create the target directory (`build_ARCH/root/` by default) and the target filesystem skeleton. This directory will contain the final root filesystem. To setup it up, it first deletes it, then it uncompress the `target/default/skel.tar.gz` file to create the main subdirectories and symlinks, copies the skeleton available in `target/default/target_skeleton` and then removes useless `CVS/` directories.
- Make the `TARGETS` dependency. This is where all the job is done : all `Makefile.in` files "subscribe" targets into this global variable, so that the needed tools gets compiled.

Using the uClibc toolchain

You may want to compile your own programs or other software that are not packaged in Buildroot. In order to do this, you can use the toolchain that was generated by Buildroot.

The toolchain generated by Buildroot by default is located in `build_ARCH/staging_dir/`. The simplest way to use it is to add `build_ARCH/staging_dir/bin/` to your PATH environnement variable, and then to use `arch-linux-gcc`, `arch-linux-objdump`, `arch-linux-ld`, etc.

For example, you may add the following to your `.bashrc` (considering you're building for the MIPS architecture and that Buildroot is located in `~/buildroot/`) :

```
export PATH=\$PATH:~/buildroot/build_mips/bin/
```

Then you can simply do :

```
mips-linux-gcc -o foo foo.c
```

Important : do not try to move the toolchain to an other directory, it won't work. There are some hard-coded paths in the gcc configuration. If the default toolchain directory doesn't suit your needs, please refer to the *Using the uClibc toolchain outside of buildroot* section.

Using the uClibc toolchain outside of buildroot

By default, the cross-compilation toolchain is generated inside `build_ARCH/staging_dir/`. But sometimes, it may be useful to install it somewhere else, so that it can be used to compile other programs or by other users. Moving the `build_ARCH/staging_dir/` directory elsewhere is not possible, because they are some hardcoded paths in the toolchain configuration. If you want to use the generated toolchain for other purposes, you can configure Buildroot to generate it elsewhere using the option of the configuration tool : Build options -> Toolchain and header file location, which defaults to `$(BUILD_DIR)/staging_dir/`.

Location of downloaded packages

It might be useful to know that the various tarballs that are downloaded by the Makefiles are all stored in the `DL_DIR` which by default is the `dl` directory. It's useful for example if you want to keep a complete version of Buildroot which is know to be working with the associated tarballs. This will allow you to regenerate the toolchain and the target filesystem with exactly the same versions.

Extending Buildroot with more software

This section will only consider the case in which you want to add user-space software.

Package directory

First of all, create a directory under the package directory for your software, for example `foo`.

`Config.in` file

Then, create a file named `Config.in`. This file will contain the portion of options description related to our foo software that will be used and displayed in the configuration tool. It should basically contain :

```
config BR2_PACKAGE_FOO
        bool "foo"
        default n
        help This is a comment that explains what foo is.
```

Of course, you can add other options to configure particular things in your software.

`Makefile.in` file

Then, write a `Makefile.in` file. Basically, this is a very short *Makefile* that adds the name of the software to the list of `TARGETS` that Buildroot will generate. In fact, the name of the software is the the identifier of the target inside the real Makefile that will do everything (download, compile, install), and that we study below. Back to `Makefile.in`, here is an example :

```
ifeq ($(strip $(BR2_PACKAGE_FOO)),y)
TARGETS+=foo
endif
```

As you can see, this short `Makefile` simply adds the target foo to the list of targets handled by Buildroot if software foo was selected using the configuration tool.

The real Makefile

Finally, here's the hardest part. Create a file named `foo.mk`. It will contain the *Makefile* rules that are in charge of downloading, configuring, compiling and installing the software. Below is an example that we will comment afterwards.

Listing A.1 – `foo.mk`

```
1  ################################################################
2  #
3  # foo
4  #
5  ################################################################
6  FOO_VERSION:=1.0
7  FOO_SOURCE:=less-$(FOO_VERSION).tar.gz
8  FOO_SITE:=http://www.foosoftware.org/downloads
9  FOO_DIR:=$(BUILD_DIR)/less-$(FOO_VERSION)
10 FOO_BINARY:=foo
11 FOO_TARGET_BINARY:=usr/bin/foo
12
13 $(DL_DIR)/$(FOO_SOURCE):
14         $(WGET) -P $(DL_DIR) $(FOO_SITE)/$(FOO_SOURCE)
15
16 $(FOO_DIR)/.source: $(DL_DIR)/$(FOO_SOURCE)
17         zcat $(DL_DIR)/$(FOO_SOURCE) | tar -C $(BUILD_DIR) $
18 (TAR_OPTIONS) -
19         touch $(FOO_DIR)/.source
20
21 $(FOO_DIR)/.configured: $(FOO_DIR)/.source
22         (cd $(FOO_DIR); \
23                 $(TARGET_CONFIGURE_OPTS) \
24                 CFLAGS="$(TARGET_CFLAGS)" \
25                 ./configure \
26                 --target=$(GNU_TARGET_NAME) \
27                 --host=$(GNU_TARGET_NAME) \
28                 --build=$(GNU_HOST_NAME) \
29                 --prefix=/usr \
30                 --sysconfdir=/etc \
31         );
32         touch $(FOO_DIR)/.configured;
33
34 $(FOO_DIR)/$(FOO_BINARY): $(FOO_DIR)/.configured
35         $(MAKE) CC=$(TARGET_CC) -C $(FOO_DIR)
36
37 $(TARGET_DIR)/$(FOO_TARGET_BINARY): $(FOO_DIR)/$(FOO_BINARY)
38         $(MAKE) prefix=$(TARGET_DIR)/usr -C $(FOO_DIR) install
39         rm -Rf $(TARGET_DIR)/usr/man
40
41 foo: uclibc ncurses $(TARGET_DIR)/$(FOO_TARGET_BINARY)
```

```
42
43 foo-source: $(DL_DIR)/$(FOO_SOURCE)
44
45 foo-clean:
46         $(MAKE) prefix=$(TARGET_DIR)/usr -C $(FOO_DIR) uninstall
47         -$(MAKE) -C $(FOO_DIR) clean
48
49 foo-dirclean:
50         rm -rf $(FOO_DIR)
```

First of all, this Makefile example works for a single binary software. For other software such as libraries or more complex stuff with multiple binaries, it should be adapted. Look at the other `*.mk` files in the package directory.

At lines 6-11, a couple of useful variables are defined :

- `FOO_VERSION` : The version of foo that should be downloaded.
- `FOO_SOURCE` : The name of the tarball of foo on the download website of FTP site. As you can see `FOO_VERSION` is used.
- `FOO_SITE` : The HTTP or FTP site from which foo archive is downloaded. It must include the complete path to the directory where `FOO_SOURCE` can be found.
- `FOO_DIR` : The directory into which the software will be configured and compiled. Basically, it's a subdirectory of `BUILD_DIR` which is created upon decompression of the tarball.
- `FOO_BINARY` : Software binary name. As said previously, this is an example for a single binary software.
- `FOO_TARGET_BINARY` : The full path of the binary inside the target filesystem.

Lines 13-14 defines a target that downloads the tarball from the remote site to the download directory (`DL_DIR`).

Lines 16-18 defines a target and associated rules that uncompress the downloaded tarball. As you can see, this target depends on the tarball file, so that the previous target (line 13-14) is called before executing the rules of the current target. Uncompressing is followed by touching a hidden file to mark the software has having been uncompressed. This trick is used everywhere in Buildroot Makefile to split steps (download, uncompress, configure, compile, install) while still having correct dependencies.

Lines 20-31 defines a target and associated rules that configures the software. It depends on the previous target (the hidden `.source` file) so that we are sure the software has been uncompressed. In order to configure it, it basically runs the well-known `./configure` script. As we may be doing cross-compilation, *target*, *host* and *build* arguments are given. The prefix is also set to `/usr`, not because the software will be installed in `/usr` on your host system, but in the target filesystem. Finally it creates a `.configured` file to mark the software as configured.

Lines 33-34 defines a target and a rule that compiles the software. This target will create the binary file in the compilation directory, and depends on the software being already configured (hence the reference to the `.configured` file). It basically runs make inside the source directory.

Lines 36-38 defines a target and associated rules that install the software inside the target filesystem. It depends on the binary file in the source directory, to make sure the software has been compiled. It uses the install target of the software Makefile by passing a prefix argument, so that the Makefile doesn't try to install the software inside host `/usr` but inside

target `/usr`. After the installation, the `/usr/man` directory inside the target filesystem is removed to save space.

Line 40 defines the main target of the software, the one referenced in the `Makefile.in` file. This targets should first of all depends on the dependecies of the software (in our example, *uclibc* and *ncurses*), and then to the final binary. This last dependency will call all previous dependencies in the right order.

Line 42 defines a simple target that only downloads the code source. This is not used during normal operation of Buildroot, but might be useful.

Lignes 44-46 define a simple target to clean the software build by calling the Makefiles with the appropriate option.

Lines 48-49 define a simple target to completely remove the directory in which the software was uncompressed, configured and compiled.

Conclusion

As you can see, adding a software to buildroot is simply a matter of writing a Makefile using an already existing example and to modify it according to the compilation process of the software.

If you package software that might be useful for other persons, don't forget to send a patch to Buildroot developers !

Annexe B

Interrupt handling on the NPP board

The way interrupts are handled on the NPP board doesn't match easily to the way the kernel Linux handles interrupts. So, the implementation of the interrupt handling code for the NPP board is a bit tricky and requires some explanations. This short document tries to explain what happens in the kernel when an interrupt is generated. As interrupt handling is split into several *levels*, the explanation is organized according to this architecture.

All references to documentation concern the RM9000 processor documentation, issue 2. This documentation is only available under NDA with PMC-Sierra corporation.

First level

When the processor receives an interrupt, it starts the corresponding handler, according to Table 110 (page 264). External interrupts have their very primary handler at `0x80000180`.

Under Linux, the function `arch/mips/kernel/traps.c:trap_init()` starts by copying the `arch/mips/kernel/genex.S:except_vec3_generic()` function to `0x80000180` using `memcpy()`. Note that this handler may be overwritten later in `trap_init()`, depending on the processor type (R4000 processors seem to have a different handler).

This function is the very first interrupt handler. It determines the cause of the interrupt by reading bit `6:2` (ExcCode) of the `CP0_CAUSE` register. See table 101 (page 251) for more informations on the `CP0_CAUSE` register. Once it has determined the cause of the interrupt (a number on 5 bits), it calls the corresponding handler using the `exception_handlers[]` array.

These handlers can be set using the `set_except_vector()` function in `arch/mips/kernel/traps.c`. The `trap_init()` function initializes a couple of exception vectors, to handle TLB exceptions, address or bus error exception, syscall exception, etc. Refer to Table 102 (page 252) to get the complete list of exception. All these handlers are named `handle_*`, and are generated using a macro at the end of `arch/mips/kernel/genex.S`.

Second level

As stated on table 102, external interrupts are all mapped to exception 0. `trap_init()` leaves this exception with the default handler (`handle_reserved`), but our code in `arch/mips/npp-board/irq.c:arch_init_irq()` calls `set_except_vector()` to register

`npp_board_handle_int()` as the handler for exception 0. This handler is implemented in the `arch/mips/npp-board/int-handler.S` file.

So, when the first-level interrupt handler determined that the ExcCode (bits 6 :2 of `CP0_CAUSE`) was 0, our `npp_board_handle_int()` function gets called.

This function reads the `CP0_CAUSE` register again to determine the cause of the external interrupts. It reads bits 23 :8 of the `CP0_CAUSE` which is a bitmap of pending interrupts. These interrupts are detailled in table 101 (page 251).

Our code in `arch/mips/npp-board/int-handler.S`: `npp_board_handle_int()` then jumps to different labels depending on the source of the interrupt. The interesting interrupts are interrupts IP2 and IP3, which are PnInt0 and PnInt1 in the RM9000 documentation. These two interrupts are directly connected to the Marvell chip, so we receive IRQs only on these two interrupt lines. Both handlers (`ll_irq0` and `ll_irq1`) of these interrupts call the `arch/mips/npp-board/irq-cic.c`: `ll_rm9k_cic_irq()` function with the interrupt number as argument (might be 2 or 3).

Interrupt IP7 is also very important. On our system, it's not PnInt5, but the timer interrupt (because TimIntDis is not set, again see table 101 for details). It's the timer interrupt that allow multitasking, so it's quite important ;-) The handler (`ll_cputimer_irq`) simply calls `arch/mips/kernel/time.c`: `ll_timer_interrupt()`, which we'll do the job.

Third level

The third level is the CIC level, with the `ll_rm9k_cic_irq()` handler.

Note that the CIC is "before" the processors. The CIC receives external interrupts, and can decided (according to a given configuration) to forward interrupts to either or both processors. This behaviour is configured in the *INTPINx* registers. By default, interrupts are forwarded to both processors.

Back to the `ll_rm9k_cic_irq()` handler. This function has to handle all pending interrupts listed in the `RM9000x2_CIC_INT_PN_STATUS_OFFSET` register (which is a bitmap of the pending IRQs). Using the `fls()` function it determines which bit is set in this bitmap, and then computes the number of the real IRQ to be called.

You might wonder what the "number of the real IRQ" is. As you understood, there are many interrupt levels, each of which transform a single interrupt to several interrupts. There is a somewhat hierarchical dispatch of the interrupts, but the Linux Kernel only has an array (`irq_desc[]` at the top of `arch/mips/kernel/irq.c`) to register all interrupts. So we need to "flatten" the hierarchical organization of interrupts into a flat interrupt numbering.

On the NPP board platform, here is the interrupt numbering :

- From `NPP_BOARD_CP0_IRQ_BASE` to `NPP_BOARD_CIC_IRQ_BASE`, that is from 0 to 15, there are `RM9000x2_CP0_IRQ_NUM` (16) interrupts, corresponding to the one described in the "Second level", and selected by the Interrupt Pending (23 :8) part of the `CP0_CAUSEq` register.
- From `NPP_BOARD_CIC_IRQ_BASE` (16) to `NPP_BOARD_MV64340_IRQ_BASE` (272), there are `RM9000x2_CIC_IRQ_NUM` (256) interrupts, corresponding to the 256 interrupts handled by the CIC. This large space is split in 8 regions of 32 interrupts. Each region correspond to a PnIntX interrupt. For more informations, see section 9.5 of the RM9000 documentation.
  Marvell IRQ's are mapped to CIC IRQ's using the `rm9k_cic_map_ext_irq()` function in `arch/mips/npp-board/irq-cic.c`. Currently, only one Marvell IRQ is map-

ped : the NPP_BOARD_MV64340_IRQ0 (16 is the CIC space, that is 32 is the global IRQ space since CIC space starts at 16). This IRQ is mapped to the first CIC interrupt. It is still unclear what this magic number 16 is, but if we change it, nothing works anymore.

– At NPP_BOARD_MV64340_IRQ_BASE (272) starts the Marvell IRQ space, which is 64 interrupts wide. To compute the number of the "real IRQ", the following formula is used :

```
irq_src =
  /* Beginning of the CIC space */
  NPP_BOARD_CIC_IRQ_BASE +

  /* Compute the offset of the 31 interrupt range
     corresponding to THIS PnINTX */
  ((irq - RM9000x2_CP0_PnINT0) << 5) +

  /* The offset inside the 32 interrupt range */
  irq_src;
```

"irq" is a parameter of the ll_rm9k_cic_irq() function. "irq_src" has been determined through fls() (see above). In the formula, (irq-RM9000x2_CP0_PnINT0) is the id of the 16 interrupts area in which the interrupt occured. Shifting it by 5 is equivalent to a multiply by 32, to get the base IRQ number of the interrupt area. Then irq_src is the offset inside this 32 interrupt area.

```
-------------------------------------------------------------------
|       | 32 int   | 32 int   | 32 int   |          |          |
|       | area 0   | area 1   | area 2   |          |          |
|       |          |          |          |          |          |
-------------------------------------------------------------------
       /<--------------------><---->
CIC_IRQ_BASE      \               irq_src
                   \
                    ((irq - RM9000x2_CP0_PnINT0) << 5)

For example, if irq=18 and irq_src=5 :
     (irq-RM9000x2_CP0_PnINT0)        = (18-16)  = 2
       ==> Area 2
     ((irq-RM9000x2_CP0_PnINT0) << 5) = (2 << 5) = 64
       ==> Area 2 offset

Then the IRQ number is 16 + 64 + 5 = 85.
```

Using this "real IRQ number", the code in arch/mips/npp-board/irq-cic.c : ll_rm9k_cic_irq() calls the ll_irq_dispatcher() function defined in arch/mips/npp-board/irq.c.

Fourth level

The ll_irq_dispatcher() function simply checks if the interrupt raised are Marvell IRQs. As Marvell is linked to the CPU through two hardware lines, they are two possible associated IRQs in the CIC space, that we find here under the names NPP_BOARD_

`MV64340_IRQ0` and `NPP_BOARD_MV64340_IRQ1`. If these interrupts are raised, then the `ll_mv64340_irq()` function of `arch/mips/kernel/irq-mv6434x.c` is called.

Fifth level

The `ll_mv64340_irq()` function of `arch/mips/kernel/irq-mv6434x.c` reads the Marvell registers `MV64340_MAIN_INTERRUPT_CAUSE_LOW` and `MV64340_MAIN_INTERRUPT_CAUSE_HIGH` to determine which interrupt occured. As these registers are bitmaps, the `fls()` function is used again to determine the IRQ number.

Finally, the `do_IRQ()` kernel-wide function (see `kernel/irq/handle.c`) is called with the final IRQ number as argument. The kernel wide system then takes the interrupt in charge, and will call the appropriate handler, that has been registered previously through the use of `request_irq()`.