Rapport de stage Ingénieur

****

Implémentation d'un ordonnanceur temps réel sur plateforme multi-cœur hétérogène

****

BELPOIS Vincent

2023

Remerciements

Je tiens à remercier Antoine Bertout, mon maitre de stage, pour m'avoir accueilli au sein du laboratoire du LIAS et pour m'avoir encadré durant ce stage. Je le remercie aussi pour la confiance qu'il m'a accordée en me laissant une autonomie dans mon travail tout en étant disponible pour répondre à mes questions et pour me guider lors de certaines difficultés. Je le remercie aussi pour m'avoir permis de participer à la vie du laboratoire notamment en réalisant un séminaire sur les travaux de mon stage où j'ai pu présenter puis échanger avec les membres du laboratoire sur mon travail.

Je remercie aussi Thomas Gaspard, doctorant au LIAS et membre du projet SHRIMP, pour avoir été disponible pour répondre à mes questions et pour m'avoir aidé à comprendre certains concepts.

Je tiens aussi à remercier l'ensemble du laboratoire du LIAS pour m'avoir accueilli durant ce stage et pour m'avoir permis de participer à la vie du laboratoire.

Je tiens aussi à dire merci aux autres stagiaires, notamment Tanguy Relo et Tom Ambroise pour leurs diverses aides tout au long de ce stage.

Table des matières

1 Introduction

J'ai pu réaliser mon stage ingénieur durant ma deuxième année d'étude à l'ISAE-ENSMA, au laboratoire du LIAS de Chasseneuil-du-Poitou. Le LIAS, ou Laboratoire d'Informatique et d'Automatique pour les Systèmes, regroupe plusieurs dizaines d'enseignants-chercheurs dans les domaines de l'automatique, le génie électrique et l'informatique. Le site de Chasseneuil-du-Poitou regroupe deux équipes, l'équipe Ingénierie des Données et des moDèles (IDD) et l'équipe Systèmes Embarqués Temps Réel (SETR).

J'ai été accueilli au sein de cette dernière afin de travailler avec Antoine Bertout, maitre de conférence et enseignant chercheur à l'Université de Poitiers, et Thomas Gaspard, ingénieur ENSMA et doctorant au LIAS, sur le projet SHRIMP. Mon stage s'intéresse à l'implémentation d'un ordonnanceur sur plateforme hétérogène[1], tandis que le reste du projet s'intéresse entre autres à la conception d'un ordonnanceur temps réel global et dynamique pour des plateformes *unrelated*.

Mon stage aura donc pour objectif d'identifier une solution permettant de programmer un ordonnanceur temps réel sur une carte de développement particulière. Il me faudra ensuite étudier les mécanismes les plus adaptés pour la migration tâches sur cette plateforme. L'objectif final sera alors d'implémenter une politique d'ordonnancement temps réel globale hétérogène naïve, voir plus évoluée en fonction de l'avancement des travaux de thèse.

2 Contexte

Les systèmes avioniques sont habituellement des ensembles complexes reposant sur des architectures anciennes de par le niveau de certification exigé. Par conséquent, les architectures plus récentes nécessitent une phase d'étude afin d'en évaluer l'intérêt. Dans ce contexte, le projet SHRIMP a pour but d'étudier l'implémentation d'un algorithme d'ordonnancement sur une plateforme hétérogène.

2.1 Ordonnancement

L'ordonnancement est l'étude de l'affectation des ressources à des tâches. Dans le cadre de mon stage, les ressources sont des processeurs et les tâches sont des programmes. L'ordonnancement consiste donc à affecter des processeurs à des programmes ou des tâches.

2.1.1 Généralités sur l'ordonnancement

Dans les modèles théoriques d'ordonnancement, l'exécution d'une tâche τ est définie par un triplet (C, T, D) où C est le pire temps d'exécution (*Worst Case Execution Time* ou WCET), T est la période et D est l'échéance. Le WCET est le temps d'exécution maximal d'une tâche. La période est le temps entre deux exécutions d'une tâche. L'échéance est la date limite à laquelle une tâche doit être exécutée. On parlera alors de tâche périodique si T est constant, de tâche apériodiques si les tâches démarrent à intervalles irréguliers, et de tâches sporadiques lorsque une tâche est apériodique mais possède une contrainte temporelle minimum sur le temps entre deux exécutions.

On pourra par la suite représenter l'ordonnancement temps réel par un diagramme de Gantt. Un diagramme de Gantt est un diagramme qui représente l'utilisation des ressources en fonction du temps. On peut alors représenter l'exécution des tâches au cours du temps, en représentant leur réveil par une fleche montante, leur exécution par un rectangle, leur échéance ou *deadline* par une fleche vers le bas, et la fin de leur exécution par un "T".

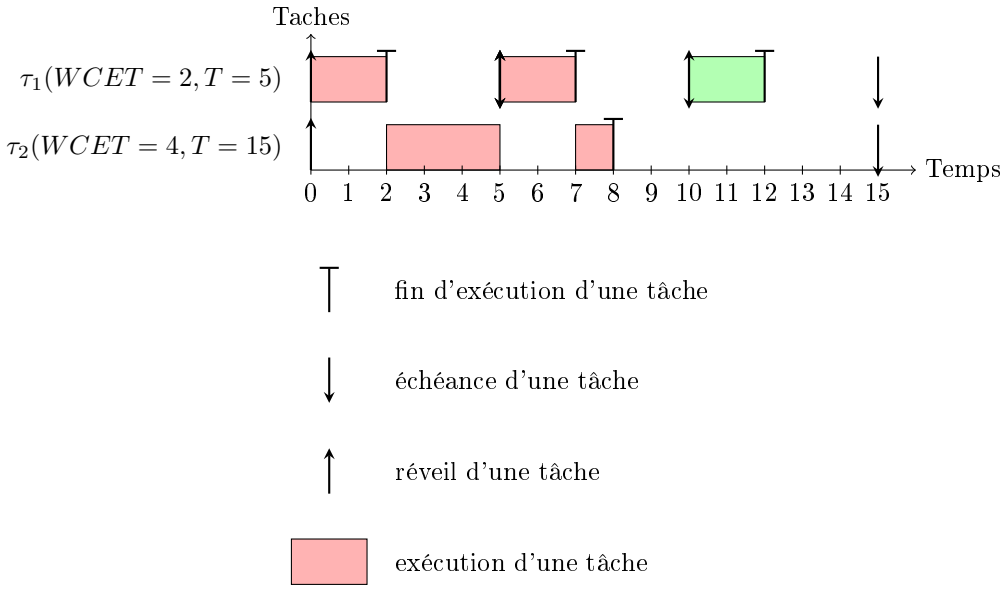

FIGURE 1 – Diagramme de Gantt de l'ordonnancement de deux taches et légende

On représentera aussi le processeur sur lequel la tâche s'exécute par une couleur de rectangle propre à chaque processeur. On a par exemple représenté le processeur 1 par un rectangle rouge et le processeur 2 par un rectangle vert sur la figure précédente.

On parlera alors de migration de tâche si une tâche change de processeur au cours de son exécution. On parlera de préemption si une tâche est interrompue par une autre tâche.

Si l'on veut avoir des tâches dont le réveil n'est pas simultané on parlera alors de tâches avec *offset*. Un *offset* est un délai entre le début de l'ordonnancement et le réveil de la tâche. Par exemple, une tâche avec un *offset* de $2ms$ se réveillera deux millisecondes après les tâches qui ont un *offset* de $0ms$.

2.1.2 Ordonnancement temps réel

Dans le cas de l'ordonnancement temps réel, les tâches sont des tâches temps réel, c'est à dire des tâches qui possèdent une contrainte temporelle. Cette contrainte peut être une échéance ou un délai maximal d'exécution. L'ordonnancement temps réel est donc l'étude de l'affectation des ressources tout en garantissant le respect des contraintes temporelles des tâches.

On peut alors définir plusieurs catégories de temps réel :

— Le temps réel dur : les contraintes temporelles doivent être respectées sous peine de défaillance du système

— Le temps réel mou ou souple : les contraintes temporelles doivent être respectées mais une violation de ces contraintes n'entraîne pas de défaillance du système

— Le temps réel ferme : les contraintes temporelles doivent être respectées mais une violation de ces contraintes entraîne une dégradation des performances du système

Si on l'on note un ensemble de tâches $\tau_1, \tau_2, \tau_3, ...$ on peut alors définir un système de tâches τ comme un ensemble de tâches τ_i avec $i \in \mathbb{N}$. On peut alors définir un système de tâches ordonnançable comme un système de tâches pour lequel il existe un ordonnancement qui permet de respecter toutes les contraintes temporelles des tâches. C'est a dire, un ensemble de réveil, de préemptions et de migrations qui permettent de respecter les contraintes temporelles des tâches.

2.1.3 Différents types d'ordonnancement

Il existe de multiples catégories d'ordonnanceurs : monoprocesseurs ou multiprocesseurs, préemptifs ou non préemptifs, globaux ou partitionnés, hors ligne ou en ligne, etc. Dans le cadre de mon stage, je me suis intéressé aux ordonnanceurs temps réel globaux et en ligne. L'aspect temps réel étant mentionné précédemment, je vais donc définir les termes globaux et en ligne.

Un ordonnanceur est dit global si il peut affecter une tâche à n'importe quel processeur. Un ordonnanceur hors ligne consiste à définir une table de séquençage en amont de la mise en route du système. Cette table de séquençage est alors utilisée par l'ordonnanceur pour ordonnancer les tâches. Un ordonnanceur en ligne consiste à ordonnancer les tâches au fur et à mesure de leur arrivée. Cela est plus complexe qu'un ordonnanceur hors ligne car il doit ordonnancer les tâches sans connaître les tâches qui vont arriver dans le futur. Cependant, cela permet de réagir plus rapidement aux événements.

2.2 Différentes plateformes

Il existe plusieurs types de plateformes, c'est a dire plusieurs manières d'arranger les processeurs. On peut les classer en quatre catégories : identique, uniforme, non uniforme et cohérente.

2.2.1 Plateforme identique

Dans une plateforme identique, tous les processeurs sont identiques. Ils possèdent la même vitesse d'exécution, la même mémoire cache, le même jeu d'instructions, etc. C'est la plateforme la plus simple à étudier et la plus documenté dans la literature scientifique.

2.2.2 Plateforme uniforme

Dans une plateforme uniforme, chaque processeur possède une vitesse d'exécution différente. Dans ce cas, un processeur de vitesse 2 exécute toutes les tâches deux fois plus vite qu'un processeur de vitesse 1. C'est une plateforme plus complexe à étudier que la plateforme identique mais moins complexe que la plateforme non uniforme. Un exemple réel d'une telle plateforme est une plateforme identique dans laquelle certains processeurs ont vu leur fréquence changé : les processeurs ayant une plus grande fréquence d'exécution sont alors des processeurs plus rapides tandis que les processeurs ayant une plus petite fréquence d'exécution sont des processeurs plus lents.

2.2.3 Plateforme non uniforme ou *unrelated*

Dans une plateforme non uniforme, chaque processeur possède une vitesse d'exécution différente. La vitesse d'exécution d'un processeur n'est pas un multiple de la vitesse d'exécution des autres processeurs. C'est la plateforme la plus complexe à étudier. Il peut alors y avoir une affinité tâche/processeur : le processeur P_1 exécute la tâche T_1 plus vite que le processeur P_2 mais le processeur P_2 exécute la tâche T_2 plus vite que le processeur P_1. Cela est une plateforme très complexe à étudier.

2.2.4 Plateforme cohérente

Dans une plateforme cohérente, chaque processeur possède une vitesse d'exécution différente qui peut être ordonné. Par exemple, le processeur P_1 exécute toutes les tâches plus vite que le processeur P_2 qui exécute toutes les tâches plus vite que le processeur P_3. Cependant, pour certaines tâches, il est concevable sous ce modèle de plateforme que les processeurs exécutent des tâches a des vitesses identiques. C'est une plateforme plus complexe à étudier que la plateforme uniforme mais moins complexe que la plateforme non uniforme.

2.3 Projet SHRIMP

On peut alors regrouper les plateformes uniformes, non uniformes et cohérentes sous le terme de plateformes hétérogènes. Le projet SHRIMP s'intéresse à l'ordonnancement temps réel sur des plateformes hétérogènes. En-effet, la démocratisation des systèmes sur puce (SoC) a permis l'émergence de plateformes hétérogènes. Par exemple, un SoC peut regrouper des clusters de processeurs, un processeur graphique, un processeur réseau. Ces SoC sont de plus en plus utilisés dans les systèmes embarqués. Par exemple, les smartphones sont des systèmes embarqués qui utilisent des SoC.

Mon stage s'intéresse alors à l'étude de l'implémentation d'ordonnanceurs temps réel sur des plateformes hétérogènes. En-effet, les ordonnanceurs temps réel existants sont souvent développés pour des plateformes homogènes. Cependant, les plateformes hétérogènes sont de plus en plus utilisées dans les systèmes embarqués. Les travaux de thèse qui sont réalisés par Thomas GASPARD s'intéressent a concevoir et étudier théoriquement des ordonnanceurs temps réel qui utilisent les caractéristiques des plateformes hétérogènes. J'ai alors pour but d'étudier l'implémentation d'ordonnanceurs temps réel sur une plateformes hétérogènes afin de faciliter l'évaluation des travaux de thèse sur une plateforme réelle.

3 Systèmes d'exploitation compatibles avec la carte ROCK960

Afin d'étudier les systèmes d'exploitation temps réel compatibles avec la carte de développement qui m'a été fournie, je me suis d'abord familiarisé avec celle-ci en installant des OS fournis par le fabricant avant d'étudier la compatibilité avec des systèmes plus complexes.

3.1 Présentation de la carte de développement

Le stage s'intéressant à l'implémentation d'un algorithme d'ordonnancement sur une plateforme hétérogène, une carte possédant un tel processeur est mis à ma disposition. Cette carte se nomme ROCK960 et est fabriquée par l'entreprise *96Boards*. Cette carte de développement contient de nombreuses interfaces, mais nous nous contenteront d'utiliser l'interface série TTL. Nous nous connecterons à cette dernière via un convertisseur USB vers TTL. Cela permettra d'interfacer via un terminal qui fonctionnera avec une liaison série.

3.1.1 Le processeur RK3399

Au centre de la carte est un SOC Rockchip RK3399. Ce processeur contient deux type de cœurs, ou processeurs. Deux d'entre eux sont des processeurs Cortex-A72 et les quatre autres sont des processeurs Cortex-A53. Ces 6 processeurs utilisent le même jeu d'instruction : ARMv8-A 64-bit. Cela sera important par la suite afin de faciliter la migration de tâches entre les processeurs, en effet si les jeux d'instructions des processeurs étaient différents, plusieurs copies du code compilé devraient exister tout en maintenant un lien d'équivalence entre les deux codes. Cela est bien au delà de la portée de mon stage mais serait un point intéressant à explorer.

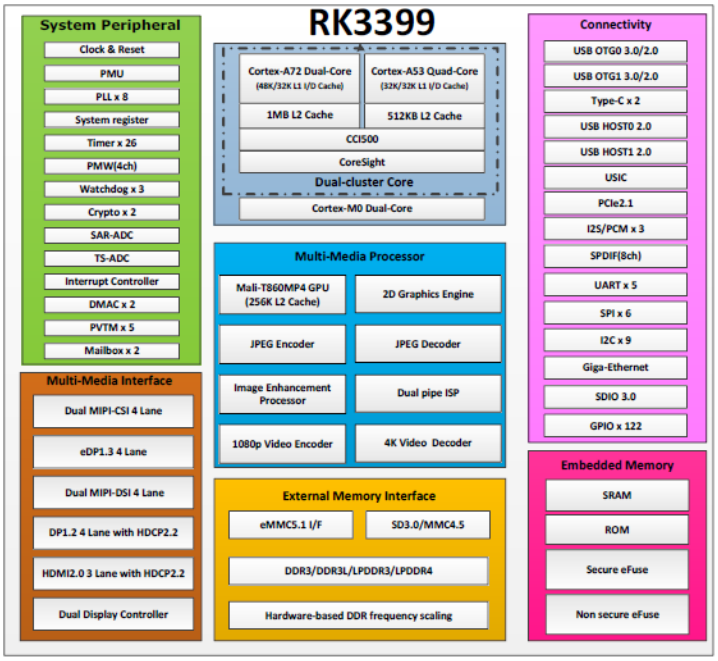

FIGURE 2 – Architecture du processeur RK3399

Le SOC RK3399 contient bien d'autres composants et peut interfacer avec de nombreux périphériques (écran HDMI, USB, caméra MPI-CSI, SPI, UART, I2C, etc...) comme le montre la figure 2. Ce diagramme nous montre aussi que les deux cluster de processeurs ne partagent ni les caches L1 ni les caches L2, mais sont interconnectés par une interface CCI-500 qui, selon le site des développeurs ARM, permet la cohérence des caches des deux cluster.

3.1.2 Interfacer avec la carte

Comme mentionné précédement, la carte possède une sortie HDMI, mais nous nous contenterons d'utiliser l'interface série TTL. Nous nous connecterons à cette dernière via un convertisseur USB vers TTL. Cela permettra d'interfacer via un terminal qui fonctionnera avec une liaison série. J'ai utilisé l'utiitaire en ligne de commande *minicom* pour me connecter à la carte. Pour que cela

fonctionne, il faut configurer le *baudrate* à une vitesse de 1M bauds, 8 bits de données, 1 bit de parité et 1 bit de stop (8N1). La figure 3 montre le terminal série de *minicom* connecté à la carte.

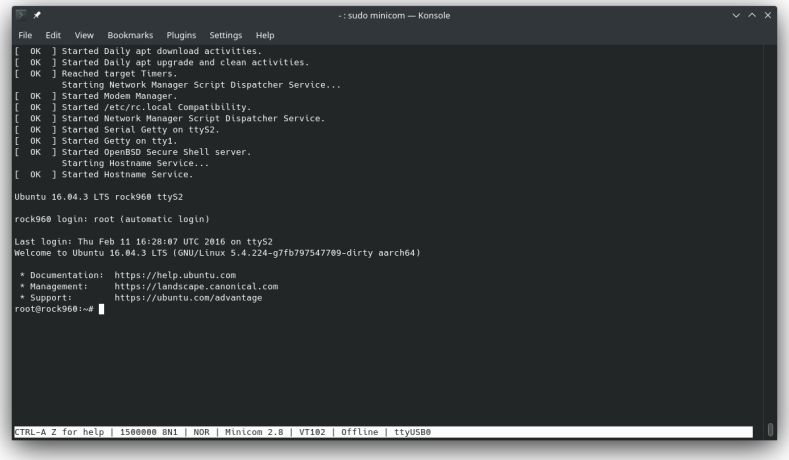

FIGURE 3 – Terminal série via minicom connecté à la carte

3.2 Installation d'un système d'exploitation

Le code que l'on veut exécuter sur la carte de développement doit être compilé pour celle-ci, puis placé sur un support de stockage. Ce code peut être un simple programme ou bien un système d'exploitation complet. J'ai premièrement installé des images pré-compilées de Linux, puis j'ai compilé moi-même le noyau Linux depuis son code source en réalisant une compilation croisée.

3.2.1 Installation d'une image pré-compilée

Dans un premier temps, pour tester l'installation de linux sur la carte de développement, j'ai utilisé une image de la distribution Ubuntu fournie par le fabriquant *96Boards* disponible sur leur site. Cette image se présente sous la forme d'une archive au format `.tar.gz`. Elle contient à la fois le bootloader, le noyau Linux, et le système de fichiers. Cette image (`system.img`) peut alors être gravée (ou *flashée*) sur une carte micro SD.

Depuis un terminal, en se déplaçant dans le dossier de l'archive extraite, on exécute la commande suivante :

```
$ sudo dd if=system.img of=/dev/XXX bs=4M oflag=sync status=noxfer
```

Listing 1 – Téléversement de l'image sur la carte microSD

Cette commande permet de copier le contenu de l'image sur la carte micro SD. Le paramètre `if` permet de spécifier le fichier source, ici l'image de la carte. Le paramètre `of` permet de spécifier le fichier de destination, ici la carte micro SD. Le paramètre `bs` permet de spécifier la taille des blocs de lecture et d'écriture. Le paramètre `oflag` permet de spécifier des options de copie, ici `sync` permet de synchroniser les entrées et sorties. Le paramètre `status` permet de spécifier le niveau de détail des informations affichées, ici `noxfer` permet de ne pas afficher le nombre de blocs transférés.

3.2.2 Compilation de Linux depuis le code source

Afin d'utiliser une version de Linux différente de la version précompilé par le fabriquant de la carte de développement, il faut premièrement se procurer le code source du noyau Linux. Celui-ci est disponible sur un dépôt de code git hébergé par *GitHub*. Il est disponible a l'adresse `https://github.com/torvalds/linux`, sous le profil du créateur de Linux : Linus Torvalds. Durant mon stage, j'étais libre d'utiliser le logiciel de gestion de version de mon choix, j'ai donc principalement utilisé *git* en ligne de commande et j'ai parfois utilisé un client git nommé *GitKraken* afin d'explorer plus facilement les anciens *commits* de certains projets comme LITMUS^{RT}.

Une fois le code source du noyau téléchargé, et en se déplaçant dans le dossier `linux` depuis un terminal, on peut alors procéder à la compilation. Pour mes premiers essais, j'ai décidé de compiler Linux pour une machine virtuelle que je fais tourner sur ma machine de travail.

Le noyau Linux est un programme ayant une compilation basée sur la configuration : cela signifie que certaines parties du code peuvent être rajoutées ou omises par le simple biais d'un fichier de configuration. Cette configuration se présente sous la forme d'un fichier `.config` qui doit être créé à la racine du noyau. Pour créer ce fichier, des utilitaires sont mis a notre disposition dans le noyau :

— `make defconfig` : cet outil est utilisé pour générer une configuration par défaut. Ici l'architecture de la machine qui réalise la compilation est sélectionnée. Dans mon cas, éxécuter cette commande crée un fichier de configuration basé sur la config `'x86_64_defconfig'`.

— `make menuconfig` : cet outil permet d'éditer la configuration actuelle du fichier `.config` via une interface graphique. Ce menu permet aussi de rechercher des paramètres, de voir leur description et d'enregistrer différentes configurations.

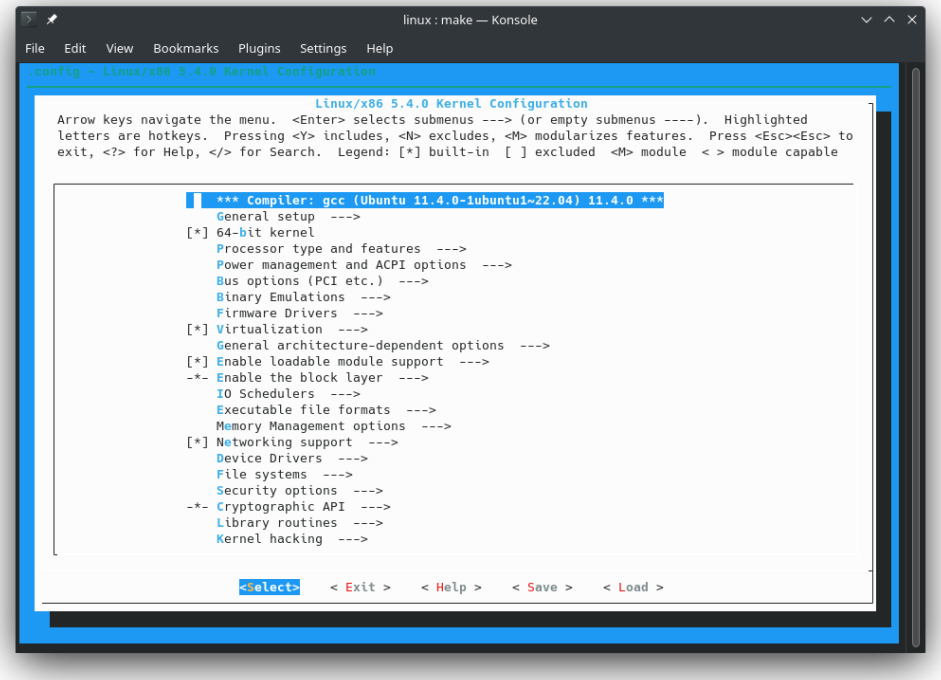

FIGURE 4 – Interface de configuration du noyau

Le menu de `make menuconfig` s'appui sur les fichiers `Kconfig` présents dans les dossiers du noyau. Ces fichiers contiennent les différentes options de configuration du noyau. Par exemple, le fichier `Kconfig` du dossier `arch/x86/Kconfig` contient les options de configuration spécifiques à l'architecture `x86`. Ces fichiers sont écrits dans un langage propre à Linux, et sont ensuite compilés en un fichier `Kconfig` binaire. Ce fichier binaire est ensuite utilisé par `make menuconfig` pour afficher les différentes options de configuration.

Une fois la configuration créée, nous pouvons passer à la compilation du noyau. Linux utilise l'utilitaire de compilation *GNU Make* : il permet l'automatisation de la compilation, la gestion des dépendances et gère la personnalisation de la compilation de chaque dossier. Ces règles sont alors dictées par des fichiers `MakeFile` présents dans chaque dossier contenant des fichiers à compiler du projet.

Il est à noter que chaque distribution Linux possède un ensemble différent de programmes préinstallés, il faudra alors peut-être installer des programmes nécessaires a la compilation. Par exemple, installer `libelf-dev`, "une bibliothèque partagée qui permet de lire et écrire des fichiers ELF à un niveau élevé" [1].

```
1 $ make -j16
```

Listing 2 – Compilation sur plusieurs processeurs

1. d'après la description sur packages.debian.org/fr/sid/libelf-dev

Le paramètre `-j 16` signifie que l'on veux exécuter la compilation avec 16 tâches en parallèle. Il est recommandé d'utiliser comme nombre de tâches le double du nombre de processeurs dans l'ordinateur qui réalise la compilation.

Par la suite, il sera parfois nécessaires de changer la version de Linux que je compile afin de tester si celle-ci fonctionne. Sur le dépôt de code de Linux, les différentes versions sont stockées sous forme d'un certain commit qui a été tagué afin de le retrouver. On peut alors changer de version en revenant à ce commit grâce à la commande `checkout` de git :

```
$ git checkout v5.4
```

Listing 3 – Retour sur un commit tagé

On peut avoir la liste de ces commits tagués de la manière suivante :

```
$ git tag -l
```

Listing 4 – Comment lister les tags

On aura alors l'ensemble des tags de tout le dépôt de code, et on peut filtrer ces résultats, avec `grep` par exemple, si l'on veut retrouver une version particulière.

Bien que j'étais déjà familier avec git, cela m'a pris un certain temps de comprendre comment ce changement de version s'effectuait. La nuance que l'on ne changeait pas de branche dans le dépôt, mais que l'on revenait simplement au commit correspondant à la version était la plus compliquée à comprendre. Tout au long de ce stage j'ai pu utiliser git afin d'explorer comment certains projets ont été construits en remontant l'historique de leurs commits, mais j'ai pu aussi utiliser git pour gérer le stockage du code que j'ai développé, qu'ils soient des outils ou des modifications du noyau.

3.2.3 Compilation croisée

Nous ne pouvons malheureusement pas compiler directement le noyau Linux pour la carte de développement du fait de la différence de jeu d'instruction. Il faut donc réaliser une *cross compilation* ou compilation croisée : c'est le fait de compiler un programme sur une architecture qui n'est pas celle de l'architecture cible.

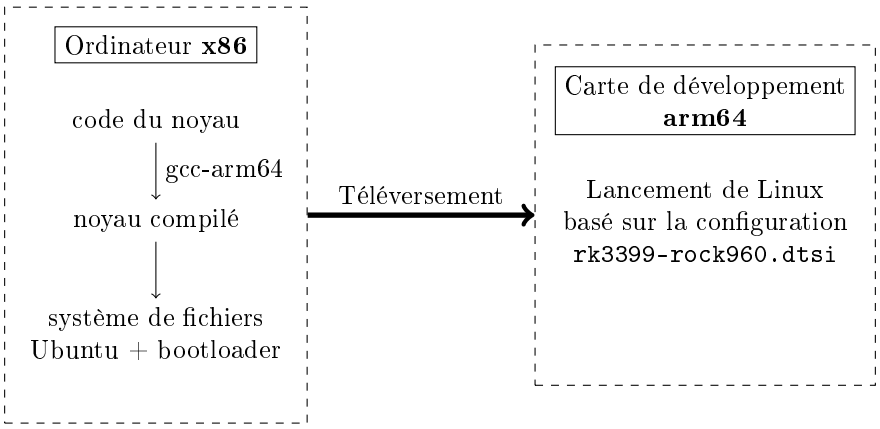

FIGURE 5 – Compilation croisée du noyau Linux

Dans notre cas, l'ordinateur qui réalisera la compilation a un jeu d'instructions `x86` tandis que la carte de développement a un jeu d'instructions `arm64`. Lors de la compilation du noyau Linux, on doit alors fixer des variables d'environnement comme l'architecture cible et le chemin vers la *toolchain* à utiliser :

```
ARCH="arm64"
CROSS_COMPILE="../toolchain/bin/aarch64-linux-gnu-"
```

Listing 5 – Variables pour la compilation croisée du noyau Linux

Ces variables seront lues par le fichier `Makefile` principal du noyau. Par exemple, le compilateur qui sera utilisé est `aarch64-linux-gnu-gcc` et fait partie de la *toolchain* que j'ai pu me procurer sur le site de *Linaro*[2]. Cette *toolchain* contient entre autres un compilateur qui s'exécutera sur

2. releases.linaro.org/components/toolchain/binaries/latest-7/aarch64-linux-gnu/gcc-*-x86_64_aarch64-linux-gnu.tar.xz

une machine `x86` et compilera vers une architecture `arm64`. Cette toolchain contient aussi des bibliothèques propres a l'architecture cible, comme `arm_neon.h` (librairie qui permet d'utiliser les unités de calcul NEON sous les architectures ARM compatibles qui permettent le calcul en parallèle grâce aux instructions SIMD).

Le noyau linux est ainsi compilé par la commande :

```
1    make Image dtbs -j16
```

Listing 6 – Compilation croisée du noyau Linux

Cela produit entre autres un fichier `Image` dans le dossier `litmus-rt/arch/arm64/boot`. Ce fichier est le noyau Linux compilé, non compressé, et contenant les modules compilés en tant que `built-in-modules` mais ne contenant pas les modules externes. Ces derniers peuvent être compilés séparément et installés par une simple copie sur le système de notre choix.

Une fois le noyau étant compilé, on peut le flasher sur la carte microSD que lira la carte de développement à son démarrage. Cependant, il faut joindre ce noyau a plusieurs autres programmes afin d'avoir un système d'exploitation utilisable :

— un *bootloader*, dans notre cas `u-boot`
— un système de fichiers et des programmes utilitaires, dans mon cas j'ai choisi une image minimale de Ubuntu[3]

Le *bootloader* est un court programme qui est chargé d'amorcer le système d'exploitation principal. Il est stocké dans une mémoire non volatile et est exécuté au démarrage de la carte de développement. Il est alors chargé de charger le noyau Linux et de lui passer la main. Dans notre cas, le fabriquant de la carte de développement fourni un *bootloader* nommé `u-boot` qui est déjà compilé et qui est disponible sur leur dépôt de code. Il est alors possible de le télécharger et de le flasher sur la carte microSD.

J'ai par la suite réalisé des scripts bash afin d'accélérer le processus de compilation et de flashage du noyau Linux. Je me suis aussi rendu compte par la suite que je pouvonsais simplement suprimer le fichier `Image` présent sur la carte micro SD et le remplacer par celui nouvellement compilé. Cela à pu accélérer le processus de flashage du noyau Linux d'une dizaine de minute, à moins d'une minute. Avant cela, il fallait flasher l'ensemble de la carte micro SD, ce qui prenait beaucoup plus de temps. Cela m'a alors permis de tester des corrections de bugs beaucoup plus rapidemement et accélérer le développement.

3.3 Etude des versions de Linux compatibles

3.3.1 Comment Linux gère le suport d'un processeur

Pour gérer la compatibilité avec un processeur, le *bootloader* charge au démarrage de Linux le fichier *Device Tree* qui contient les informations sur le matériel présent sur la carte. Ce fichier est ensuite utilisé par le noyau Linux pour initialiser le matériel. Dans notre cas, le fichier `rock960-rk3399.dts` charge le fichier `rk3399.dtsi` qui contient les informations sur le processeur. On peut y trouver les informations sur la connectique, les périphériques, les contrôleurs, les bus, etc. La partie nous intéressant est celle sur la structure des processeurs qui se trouve dans le fichier `rk3399.dtsi`. On y trouve les informations sur les différents cœurs du processeur, leur fréquence, leur cache. C'est vers la fin de mon stage que j'ai pu me rendre compte d'un oubli dans le fichier décrivant ce processeur : les caches ne sont pas décrits. Cela a des conséquences sur les algorithmes d'ordonnancement utilisant cette information pour concevoir les clusters de processeurs les plus adaptés. En effet, les caches sont utilisés pour déterminer les coûts de migration d'une tâche d'un cœur à un autre. Sans cette information, les algorithmes d'ordonnancement ne peuvent pas déterminer les coûts de migration et ne peuvent donc pas déterminer les clusters les plus adaptés.

Dans le listing 12 de l'annexe, on peut voir les modifications que j'ai apporté au fichier `rk3399.dtsi` pour ajouter les informations sur les caches. Je me suis appuyés sur les informations du *datasheet* du processeur pour ajouter ces informations. La manière d'ajouter ces informations n'étais pas documentée mais j'ai pu trouver des exemples d'autres processeurs pour m'aider à ajouter ces informations.

Ces modifications ne sont toujours pas présentes dans la version actuelle du noyau Linux, il y a alors ici une possibilité de soumettre une *pull request* pour ajouter ces informations au noyau Linux. Je me suis renseigné sur la procédure à suivre pour soumettre une *pull request* au noyau

3. ubuntu_server_16.04_arm64_rootfs_20171108.ext4

Linux et j'ai pu trouver un guide[4] expliquant la procédure à suivre. Cependant je n'ai pas eu de réponses au patch que j'ai soumis [4].

De plus, ces informations ne sont pas indispensables pour le fonctionnement de Linux sur ce processeur et ne sont donc pas une priorité pour les développeurs du noyau Linux. Il est donc possible que cette *pull request* ne soit pas acceptée : j'ai pu trouver une pull request similaire datant de plusieurs année qui a mis du temsp à être acceptée [5] alors qu'elle s'intéresse à un processeur plus répandu que le RK3399.

3.3.2 Essais de différentes versions

Le problème principal que j'ai rencontré est que le noyau LITMUS^{RT}, dont plus de détails sont donné dans la partie 4, n'est pas compatible avec les versions récentes du noyau Linux. En effet, la dernière version officielle du noyau LITMUS^{RT}est basé sur la version 4.9.30 du noyau Linux et date de Mai 2017. Cependant, la carte de développement ROCK960 n'est supportée que depuis la version 6.6 du noyau Linux comme en témoigne un commit de septembre 2018 [6]. Il est donc nécessaire de trouver une version du noyau Linux compatible avec la carte de développement et avec le noyau LITMUS^{RT}.

Comme recommandé par Antoine Bertout, les *mailling lists* de LITMUS^{RT}m'ont permis de trouver d'autres chercheurs ayant patché des versions plus récentes de Linux. Après plusieurs essaais, je me suis arrêté sur la version 5.4 du noyau Linux pour laquelle Joshua Bakita à fait le travail de patcher linux pour le rendre compatible avec LITMUS^{RT}. J'ai pu trouver son travail sur son *github* [7].

J'ai moi même essayer de réaliser ce travail pour une autre version de Linux, celle fournie par le fabriquant de la carte de développement. En effet, je n'ai jamais pu faire fonctionner le WiFi de la carte avec une autre version de Linux que celle faite par 96Boards, c'est pourquoi j'ai tenter de patcher cette version de Linux avec les commits nécessaires pour faire fonctionner LITMUS^{RT}. Cependant j'ai rencontré de nombreux problèmes, à la fois lors de la compilation, et lors du fonctionnement de ma version patché de Linux. C'est donc pourquoi j'ai arrêté ce développement pour me concentrer sur la version de Joshua Bakita. Ma compréhension du noyau Linux n'étais pas suffisante pour comprendre les problèmes que je rencontrais et j'ai donc préféré me concentrer sur la version du noyau patché qui fonctionnait.

Je n'ai pas testé le bon fonctionnement de toutes les fonctionnalités de la carte sous cette version de Linux (HDMI, USB, GPU, etc.) car je n'en avait pas besoin pour mon développement et que je n'avais pas le temps de tester toutes les fonctionnalités de la carte. Dans cette version le WiFi ne fonctionne pas, cela aurait été utile afin d'accélérer le développement lors de mon stage : téléverser les fichiers sur la carte de développement aurait été plus rapide. Cependant, le WiFi n'est pas indispensable pour le fonctionnement de la carte et je n'ai donc continuer ainsi.

Cette étape d'essais de versions et de drivers pour le WiFi étais fastidieuse et malgré l'existence d'outils pour accélérer cela comme *Yocto* ou *Buildroot*, je n'ai pas eu le temps de les utiliser. En effet, ces outils permettent de compiler un système d'exploitation Linux complet pour une carte de développement donnée. Cependant, il faut alors configurer ces outils pour qu'ils utilisent les bons drivers et les bonnes versions de Linux. Cela aurait donc nécessité de comprendre comment ces outils fonctionnent et comment les configurer pour qu'ils utilisent les bons drivers et les bonnes versions de Linux. Cela aurait été utile pour accélérer le développement mais j'ai préféré me concentrer sur le développement de l'ordonnanceur plutôt que sur la configuration de ces outils.

4. Patch envoyé à : <linux-rockchip@lists.infradead.org> et <linux-arm-kernel@lists.infradead.org>
5. https://github.com/torvalds/linux/commit/618682b350990f8f1bee718949c4b3858711eb58
6. https://github.com/torvalds/linux/commit/ffb7b25e8ac3c94f61576ca9cbfd0f16ada1be6d
7. Depôt de code de linux-5.4-litmus

4 LITMUS^{RT}

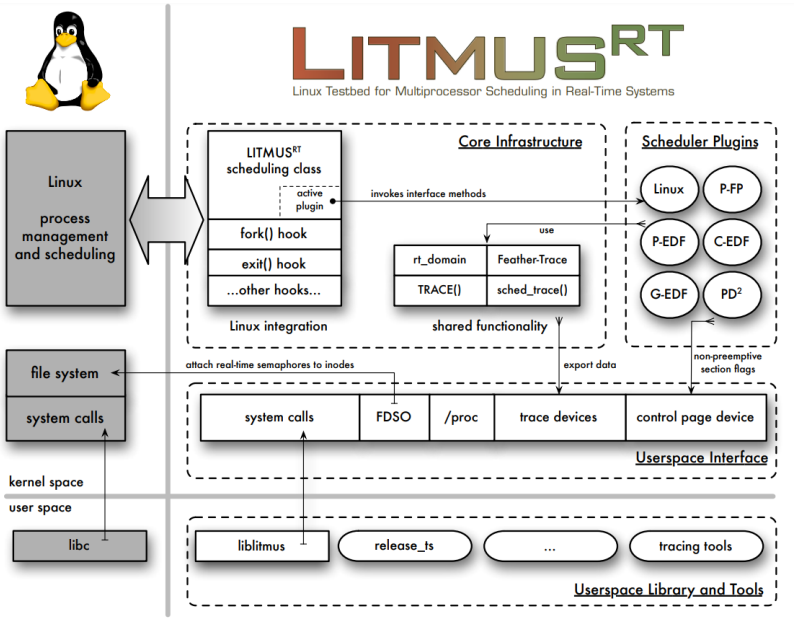

FIGURE 6 – Architecture de LITMUS^{RT}

LITMUS^{RT}est un patch au noyau Linux constitué de quatre parties :
— **LITMUS^{RT} Core** : le patch au noyau Linux qui permet d'ajouter les fonctionnalités temps réel à Linux
— plusieurs ordonnanceur
— une interface dans l'espace utilisateur
— des outils dans l'espace utilisateur tel que *feather-trace* et *liblitmus*

4.1 Présentation de LITMUS^{RT}

LITMUS^{RT}, qui signifie *Linux Testbed for Multiprocessor Scheduling in Real-Time Systems* est un moyen de développer des applications temps réel sur le noyau Linux. Il contient des modifications au noyau habituel de Linux, des interfaces utilisateurs permettant d'interagir a bas niveau avec l'ordonnancement des tâches sous Linux, ainsi qu'une infrastructure de traçage de l'exécution de l'ordonnanceur. LITMUS^{RT}à été développé par Björn B Brandenburg [3] afin de faciliter la recherche et la comparaison des algorithmes d'ordonnancement. Actuellement, beaucoup de publications utilisent LITMUS^{RT}afin de comparer différents protocoles de gestion de resources partagées par plusieurs processeurs. Mais LITMUS^{RT}est aussi utilisé pour sa facilité à être implémenté sur des plateformes récente dù au fait qu'il est construit par dessus le noyau Linux et que ce dernier est le système d'exploitation qui supporte le plus de plateformes.

Ce dernier point est principalement pourquoi nous avons choisis LITMUS^{RT}comme système d'exploitation sur lequel nous implémenterons des algorithmes d'ordonnancement pour la carte de développement ROCK960. Les autres candidats, comme FreeRTOS, étaient souvent dirigés vers les microcontrôleurs ou bien n'étaient simplement pas compatibles avec la carte de développement.

4.2 Présentation de *feather-trace*

Feather-trace [2] est outil de suivi d'événements léger conçu pour être intégré dans des applications, systèmes d'exploitation ou systèmes embarqués. Il est dans notre cas, à la fois intégré dans le noyau modifié LITMUS^{RT}, mais aussi dans les algorithmes d'ordonnancement que nous implémenterons. Il a été choisi pour sa simplicité et sa légèreté. Il permet d'enregistrer sous forme de fichier de log de multiples données de l'ordonnancement, par exemple l'arrivée d'une nouvelletâche, le début d'un nouveau job de cettetâche, la date de la fin d'exécution, et bien d'autres événements. De multiples *wrapper* des fonctions de base de *feather-trace* sont fournie dans LITMUS^{RT}afin de pouvoir log des informations supplémentaires, comme le processeur depuis lequel l'exécution du log est effectuée ou encore depuis quelle fonction l'appel est fait.

Cela a été très utile lorsque j'ai développé des nouveaux ordonnanceur sous LITMUS^{RT}afin de corriger des erreurs. Mais cet outil m'a aussi été essentiel afin de comprendre comment fonctionnait les algorithmes d'ordonnancement fournis avec LITMUS^{RT}. J'ai aussi pu comprendre comment le noyau Linux communiquait avec les ordonnanceur en activant des sorties de debug additionnels dans la configuration du noyau.

Enfin, des outils permettant d'extraire, de synthétiser ou de tracer des graphique de certaines données de ces fichiers de logs sont mis à notre disposition sur un dépôt de code présent sur github nommé *feather-trace-tools*. Voici un exemple du tracé de l'ordonnancement réel de C-EDF :

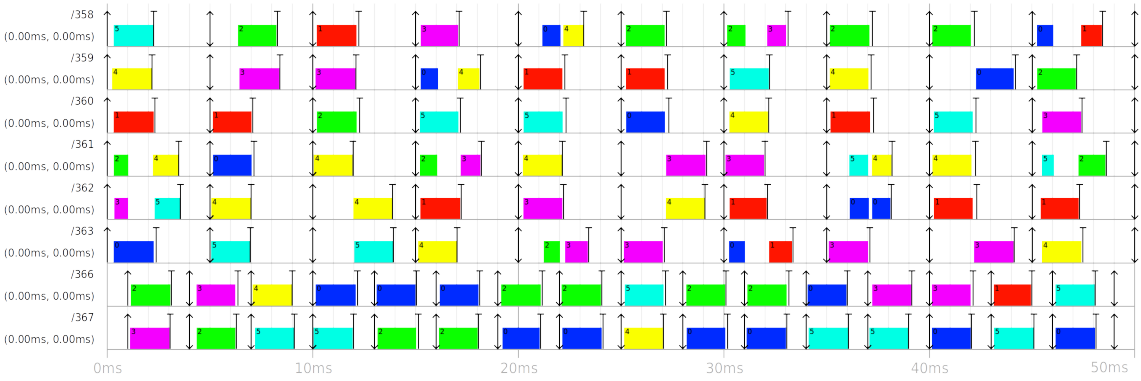

FIGURE 7 – Tracé de l'ordonnancement réel de C-EDF avec 8tâches sur les 6 processeurs

Pour cela, les temps d'exécution, les débuts et les fins de chaque jobs s'exécutant sur chaque processeur ont été enregistrés sur la carte de développement avec l'outil `st-trace-schedule` du dépôt de code mentionnée précédemment. Cet outil générer autant de fichiers que de processeurs sont présents. On peut alors tracer l'exécution réel avec cette fois ci l'outil `st-draw` en lui fournissant les fichiers générés au préalable. Ici une durée de 50ms a aussi été donnée en argument afin de limiter la durée du tracé.

4.3 Implémentation d'un ordonnanceur EDF partitionné

Le but du stage étant l'implémentation d'algorithmes d'ordonnancement sur plateforme hétérogène avec migration de tâches et de jobs entre les différent processeur, il faudra être capable de réalise des préemptions de jobs (une exécution de tâche), les migrer, assurer le traitement d'égalités et bien d'autre problèmes.

4.3.1 Algorithme considéré

On cherche alors, pour commencer, à implémenter un algorithme d'ordonnancement simple afin de se familiariser avec les méthodes et fonctions fourni par LITMUS^{RT}. J'ai donc choisi un algorithme partitionné pour la simplicité d'ordonnancement par processeur que cela offre. Un algorithme EDF (*Earliest Deadline First*) est alors choisi pour la simplicité du choix de la tâche a exécuter. Comme son nom l'indique, on choisi à chaque instant la tâche ayant l'échéance la plus proche. On nommera par la suite cet algorithme P-EDF (*Partitionned Earliest Deadline First*).

Pour montrer le fonctionnement de cet algorithme, si l'on se place sur un même processeur, on peut visualiser l'exécution de deux tâche periodiques :

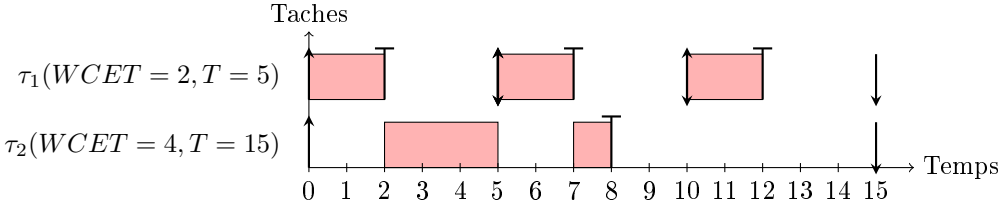

FIGURE 8 – Exemple de EDF à 2 tâches

On a ici une première tâche τ_1 avec un pire temps d'exécution (*Worst Case Execution Time*) de 2 et une période de 5, et une seconde tâche τ_2 avec un pire temps d'exécution de 4 et une période de 15. On a alors préemption de la τ_2 à $t = 5$ afin d'exécuter τ_1. Cela est dû au réveil de la tâche τ_1 (représenté par la fleche montante) et à la date d'échéance plus proche de cette dernière.

4.3.2 Implémentation

La construction d'un plugin d'ordonnancement nécessite la déclaration d'un module au sens de Linux. Pour Linux un module est un élément de code qui peut être chargé dynamiquement lors de l'exécution du système d'exploitation. Un module permet alors d'étendre les fonctionnalités du noyau, il a donc ont accès aux fonctions du noyau, à ses ressources et peut aussi réaliser des appels systèmes.

Pour que notre nouvel ordonnanceur soit reconnu par le noyau Linux modifié (LITMUS^{RT}), il faut déclarer une fonction d'initialisation :

```
#include <linux/module.h> // used for calling module_init()

static int __init init_p_edf(void)
{
    return 0; // indicates a successful initialisation
}

module_init(init_p_edf); // specify the entry point of the module
```

On peut alors enregistrer ce fichier sous le nom sched_p_edf.c pour suivre la nomenclature des autres ordonnanceurs fournis avec avec LITMUS^{RT}. Ce fichier est enregistré dans le dossier llinux /litmus. On peut alors modifier le fichier Makefile de ce dossier afin de l'ajouter au fichier à compiler :

```
    obj-y = sched_p_edf.o
```

On place notre fichier à compiler sous le mot-clé `obj-y` pour signifier que l'on veut ce module compilé et inclus lors de la compilation du noyau Linux.

Une fois le makefile modifié, la compilation de notre module sera exécutée lors de la compilation du noyau Linux à l'aide de make. La compilation du noyau est discutée dans la partie 3.2.2.

Il faut aussi déclarer un ensemble de fonctions propres à l'ordonnancement, comme pour l'admission de tâches, le réveil d'une tâche, la fin d'une tâche, le démarrage de l'ordonnanceur, etc. Voici l'ensemble des fonctions que j'ai déclaré pour mon ordonnanceur :

```
static struct sched_plugin p_edf_plugin = {
    .plugin_name          = "P-EDF",
    .schedule             = p_edf_schedule,
    .task_wake_up         = p_edf_task_resume,
    .admit_task           = p_edf_admit_task,
    .task_new             = p_edf_task_new,
    .task_exit            = p_edf_task_exit,
    .get_domain_proc_info = p_edf_get_domain_proc_info,
    .activate_plugin      = p_edf_activate_plugin,
    .deactivate_plugin    = p_edf_deactivate_plugin,
    .complete_job         = complete_job,
};
```

Listing 7 – Déclaration des fonctions de l'ordonnanceur

LITMUS^{RT}met à notre disposition un système d'abstraction pour ces fonctions afin que chaque ordonnanceur soit compatible avec les fonctions de LITMUS^{RT}.

4.3.3 Résultats et essais

Un algorithme partitionné nécessite le démarrage des tâches sur un processeur en particulier. Par exemple, j'ai ici démarré deux tâches `rtspin` avec *liblitmus* : l'une à un pire temps d'exécution de 2ms et une période de 5ms tandis que l'autre a un pire temps d'exécution de 4ms et une période de 7ms.

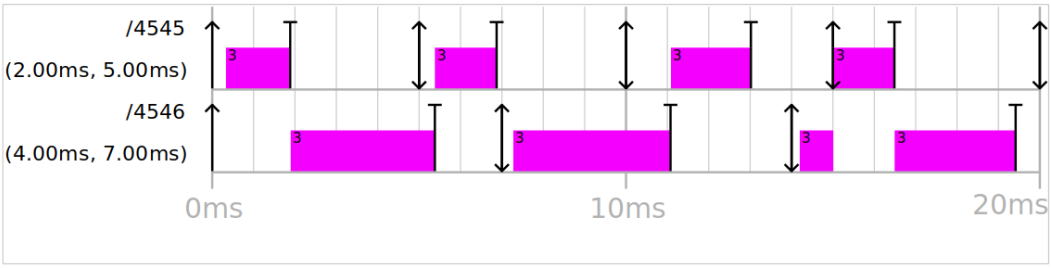

FIGURE 9 – Ordonnancement de deux tâches avec P-EDF

On peut voir qu'à $t = 5ms$, il n'y a pas préemption de la première tâche et la seconde termine son exécution. En effet, selon EDF, la deuxième tâche à une *deadline* dans 2ms, tandis que la première a une *deadline* dans 5ms : la seconde est donc à cet instant plus prioritaire que la première. Cependant, à $t = 15ms$, la seconde tâche est préemptée par la première car cette dernière se réveil et a une *deadline* dans 5ms tandis que la seconde a une *deadline* dans 6ms. On peut alors voir que la seconde tâche est préemptée à $t = 15ms$ et reprend son exécution à $t = 17ms$.

4.4 Implémentation d'un ordonnanceur RM partitionné

Comme le montre le listing de code 10, je me suis appuyé sur la librairie `litmus/edf_common.h` fournie dans LITMUS^{RT}. J'ai donc ensuite décidé d'implémenter un algorithme d'ordonnancement qui ne l'utilisait pas. J'ai donc choisi d'implémenter un algorithme RM (Rate Monotonic) partitionné. Cet algorithme est plus simple que P-EDF car il ne prend pas en compte les échéances des tâches. Il suffit alors de trier les tâches par période croissante et de les ordonnancer en fonction de leur période. Cependant, cet algorithme ne permet pas de garantir l'ordonnançabilité des tâches. En effet, il existe des ensembles de tâches qui ne sont pas ordonnançables par cet algorithme alors qu'ils le sont par P-EDF. Cependant, cet algorithme est plus simple à implémenter et permet de se familiariser avec les fonctions de LITMUS^{RT}.

4.4.1 Implémentation

Pour implémenter un algorithme RM partitionné, j'ai du réimplémenté les fonctions de `litmus/edf_common.h` pour suivre l'ordonnancement RM. Notre algorithme P-EDF faisait appel aux fonctions :

- `edf_domain_init`, qui initialise le domaine temps réel avec l'ordre qui régit la priorité des tâches
- `edf_preemption_needed`, qui vérifie si la tâche en cours d'exécutions doit être préemptée

J'ai donc implémenté les fonctions :

- `rm_domain_init`
- `rm_preemption_needed`

```
1 void rm_domain_init(rt_domain_t* rt, check_resched_needed_t resched,
2                     release_jobs_t release)
3 {
4     rt_domain_init(rt, rm_ready_order, resched, release);
5 }
```

Listing 8 – Fonction `rm_domain_init`

La complexité de cette fonction se cache derrière la nouvelle fonction d'ordre implémenté sous le nom de `rm_ready_order`. Cette fonction est passée en paramètre à `rt_domain_init` et permet d'initialiser le domaine temps réel avec l'ordre qui régit la priorité des tâches sous RM. Comme le montre le listing de code 13, cette fonction est simple et permet de trier les tâches par période croissante. En cas, d'égalité, la priorité est départagée par PID croissant (la tâche avec le PID le plus petit est la plus prioritaire). On effectue aussi d'autres vérifications sur les tâches comme leur nature (tâche temps réel ou non), si deux fois la même tâche est passée en argument, ou encore si une des tâches est `NULL` (cas ou qu'une seule tâche n'est présente).

4.4.2 Résultats et essais

En raison de la grande quantité de nouveau code, faire fonctionner cet algorithme à nécessité une plus grande phase de débogage. J'ai donc utilisé l'outil *feather-trace* pour tracer l'ordonnancement réel de cet algorithme. De cela, j'ai pu déterminer les étapes qui ne fonctionnaient pas. Par exemple, voici un exemple d'un essai avec plusieurs problèmes :

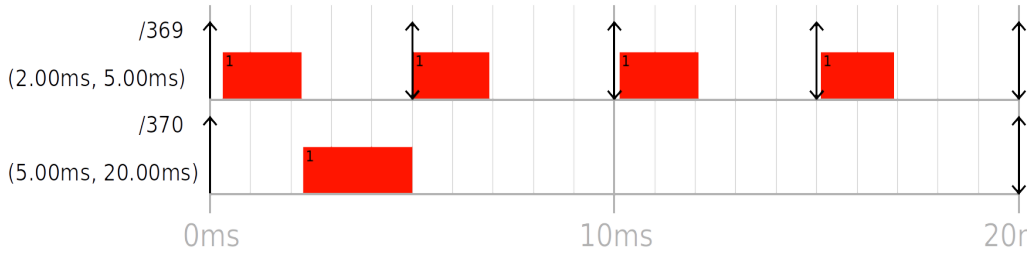

FIGURE 10 – Exemple d'ordonnancement avec défauts

Premièrement, l'ordonnanceur ne stipule pas a *feather-trace* la fin d'exécution d'une tâche. Cela peut être corrigé en appelant `sched_trace_task_completion(prev, budget_exhausted);` lors de la fin d'un job. Secondement, on peut voir que la deuxième tâche se fait préempter en $t = 5ms$ par la première tâche, plus prioritaire. Cependant, l'exécution de cette deuxième tâche ne continue pas lorsque le processeur est libre. Cette erreur était alors due à une erreur de logique dans la fonction principale d'ordonnancement dans laquelle je ne replaçais pas les tâches préemptées dans la `ready_queue`.

Après ces erreurs corrigées voici le résultat de l'ordonnancement de 2 tâches par RM toutes deux lancées sur le même processeur :

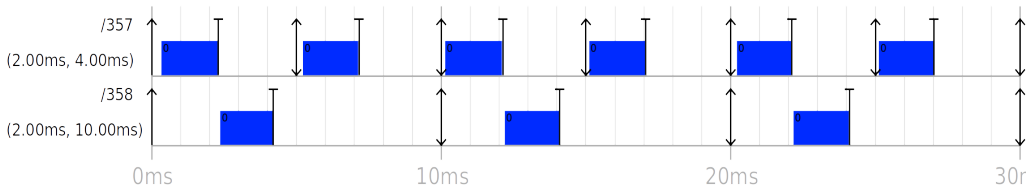

FIGURE 11 – Exemple d'ordonnancement de deux tâches par RM

On peut aussi lancer les tâches avec un *offset* afin de voir si l'ordonnancement est correct. Ce décalage est donné en tant que paramètre a la commande `rt-spin` de *liblitmus* (paramètre `-o`). Voici un exemple d'ordonnancement avec un *offset* de 1ms pour la première tâche. Les tâches ont les mêmes pires temps d'exécution et les mêmes périodes que les tâches de la figure 11.

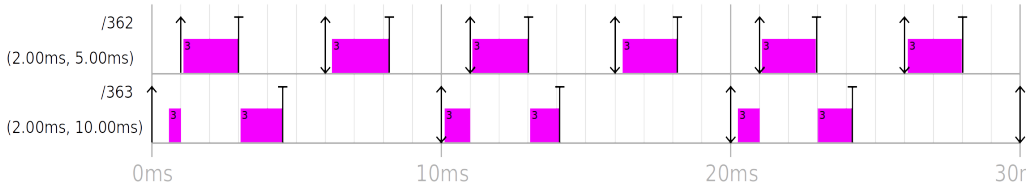

FIGURE 12 – Exemple d'ordonnancement via RM avec un offset sur une tâche

On peut aussi lancer un plus grand nombre de tâches sur une multitude de processeurs (la carte de développement en ayant 6), et on obtient le tracé de tâches suivantes :

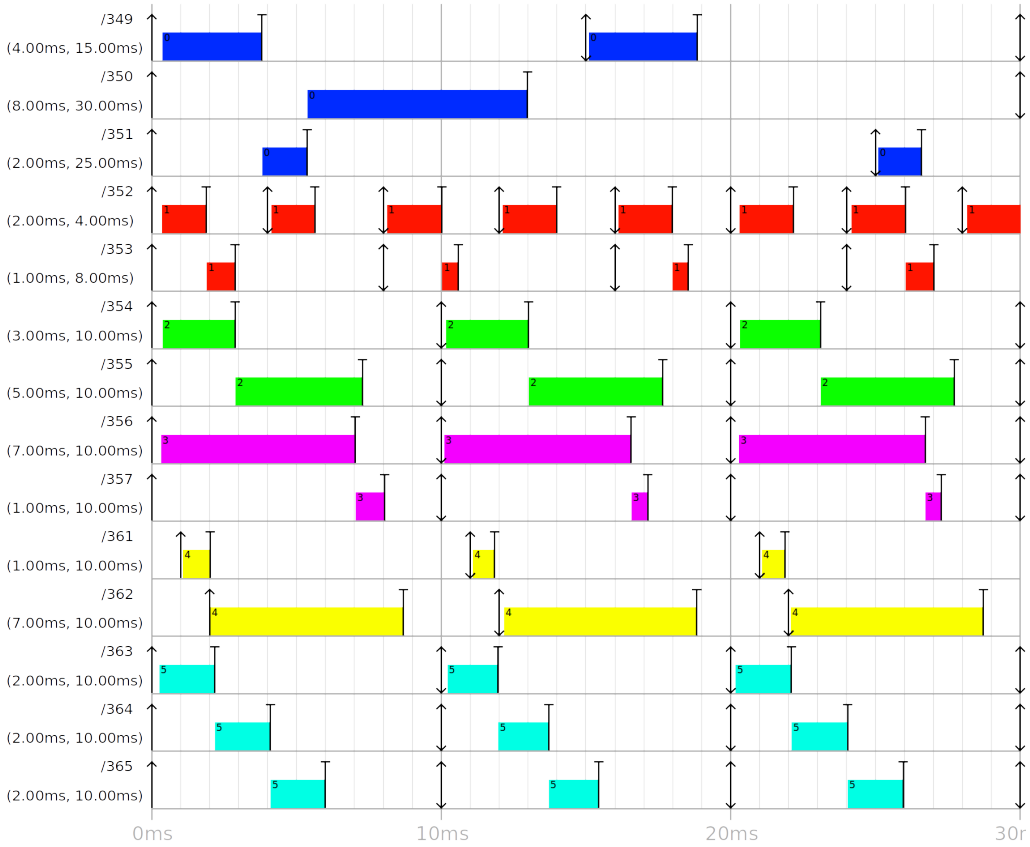

FIGURE 13 – Exemple d'ordonnancement d'un grand ensemble de tâches via RM

Enfin, on peut aussi voir ce qu'il se passe lorsque l'on cherche à ordonnancer les mêmes tâches que celles que l'on a ordonnancé avec P-EDF et que l'on peut voir dans la figure 9. Ces tâches sont non-ordonancables par RM et on obtient le tracé suivant :

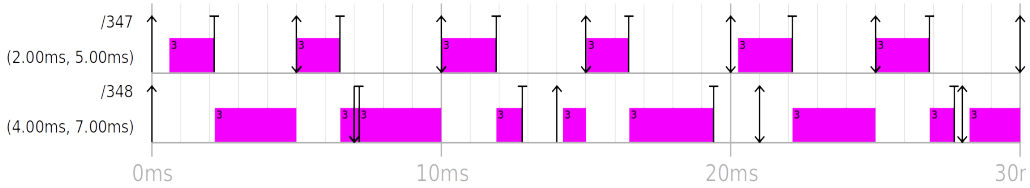

FIGURE 14 – Exemple d'ordonnancement de tâches non-ordonnançables par RM

Les dépassements des échéances montrent bien que RM est moins performant que P-EDF car il ne permet pas d'ordonnancer toutes les tâches ordonnançables par P-EDF. Cependant, il est plus simple à implémenter et m'a permis de me familiariser avec les fonctions de LITMUS^{RT}. Mais, des résultats comme le pire temps de réponse d'une tâche est plus simple à obtenir analytiquement avec RM qu'avec P-EDF. Cela est du au fait que RM est plus simple à analyser que P-EDF et c'est pourquoi il est actuellement plus utilisé dans des domaines comme l'avionique.

5 Génération et étude de tâches sur plateforme hétérogène

Lors de tout mes tests sur la carte de développement, j'ai utilisé l'outil *rtspin* de *liblitmus* afin de générer des tâches temps réel. Cet outil permet de générer des tâches avec des paramètres spécifiques, comme le pire temps d'exécution, la période, le processeur sur lequel la tâche doit s'exécuter, etc. Cependant, il ne permet pas de générer des tâches ayant des temps d'exécution différents sur différents processeurs. Cela est un problème, car cela ne permet pas de mettre en valeur la nature hétérogène de la plateforme sur laquelle nous travaillons. C'est pourquoi je me suis intéressé durant une partie de mon stage a créer de telles tâches.

5.1 Mesure de temps d'exécution

Il est important de pouvoir mesurer de manière suffisamment précise le temps d'exécution d'une tâche. Pour cela, ma première idée était d'utiliser le module `time` de Linux. Cependant, ce module ne permet pas de mesurer des temps d'exécution inférieurs à la milliseconde. Cela est dû au fait que le module `time` utilise le timer du noyau Linux qui a une précision de 1ms. Cela est bien trop imprécis pour mesurer des temps d'exécution de tâches temps réel. J'ai donc d'abord créé un script `bash` utilisant un autre temps Linux. Ce script est présent dans le listing 14 et fut utilisé pour tous mes essais préliminaires. Cependant, ce script, malgré sa plus grande précision, mesurait toujours un temps minimum : environ 6ms. Je n'ai pas pu le montrer, mais ce temps semble venir du démarrage du script, puis du démarrage du programme appelé. Il a donc été utile pour comparer des tâches entre elles, mais n'était pas assez précis pour connaître le temps d'exécution d'une tâche.

5.2 Génération de tâches répétables

Mon objectif était alors de générer des tâches qui s'exécutent à des vitesses différentes sur les différents processeurs, tout en ayant un temps d'exécution qui ne varie qu'un minimum entre deux exécutions sur un même processeur.

5.2.1 Première idée : *checksum* d'un fichier

Ma première idée était de calculer la checksum d'un fichier de petite taille. La taille du fichier permettrait alors de faire varier le temps d'exécution. Cette opération était principalement calculatoire, j'avais espoir que le temps d'exécution ne varie pas trop entre deux exécutions sur un même processeur. Cependant, cette opération semblait posséder trop d'accès à la mémoire et au stockage : deux choses que je ne voulais pas prendre en compte dans mon temps d'exécution. Comme on peut le voir sur l'exécution d'une telle tâche, sur laquelle je réalise la checksum md5 sur un fichier de code de 95KiB, J'ai donc abandonné cette idée.

5.2.2 Deuxièle idée : somme sur un grand nombre d'entiers

Ma deuxième idée était de faire une somme sur un grand nombre d'entiers. Cette opération est aussi calculatoire, mais ne possède pas d'accès mémoire. J'ai donc créé un programme qui réalise une somme sur un grand nombre d'entiers. Ce programme est présent en annexe au listing 15. Il contient aussi d'autres essais, et le choix de l'essai se fait lors de l'appel du programme. On notera par exemple que la variable sur laquelle on réalise la somme est déclarée en tant que `volatile` afin d'éviter que le compilateur optimise le code.

J'ai ensuite utilisé ce programme pour générer des tâches avec des temps d'exécution différents. Après beaucoup d'essais, et en désactivant les optimisations de compilation, j'ai réussi à obtenir des tâches avec des temps d'exécution différents. On peut alors voir le temps d'exécution en fonction de l'entier sur lequel on réalise la somme. J'ai ici réalisé le test sur deux processeurs : CPU0 qui est un processeur A53 et CPU5 qui est un des deux A72. On peut voir sur la figure 15 que le temps d'exécution est bien différent entre les deux processeurs. Cependant, on peut voir que le temps d'exécution est légèrement variable sur un même processeur. Cela est dû au fait que le processeur est partagé entre plusieurs tâches, et que le temps d'exécution d'une tâche dépend de la charge du processeur. Cela est donc un problème pour mesurer le temps d'exécution d'une tâche. Cependant, cela ne pose pas de problème pour générer des tâches avec des temps d'exécution différents. En effet, si l'on prend un entier n et que l'on réalise la somme sur les n premiers entiers, on obtient un temps d'exécution différent pour chaque valeur de n. On peut donc générer des tâches avec des temps d'exécution différents en choisissant un entier n différent pour chaque tâche. Cela est donc une solution pour générer des tâches avec des temps d'exécution différents sur différents processeurs.

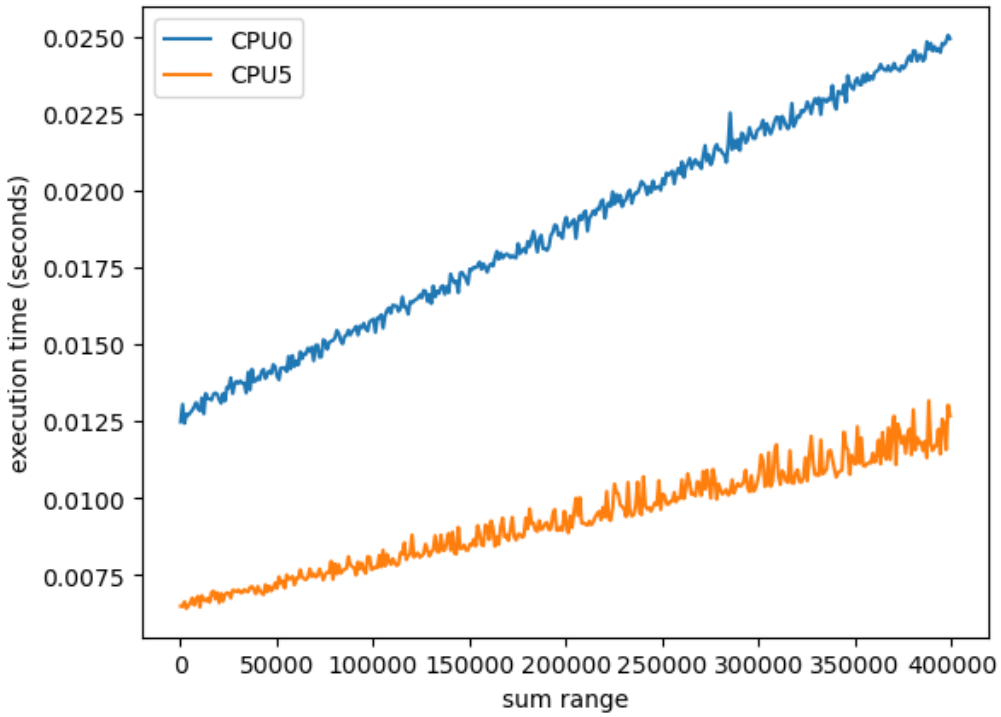

FIGURE 15 – Temps d'exécution en fonction de l'entier sur lequel on réalise la somme

On remarque aussi que le temps d'exécution semble être linéaire avec l'entier sur lequel on réalise la somme. Il m'a alors été recommandé, lors d'un séminaire où j'ai pu présenter les travaux de mon stage au laboratoire, d'étudier si cette différence de temps d'exécution pouvait être corrélée avec les différentes fréquences des processeurs. En réutilisant les données qui ont permis de tracer le graphe de la figure 15, j'ai pu obtenir les régressions linéaires suivantes pour les deux processeurs :

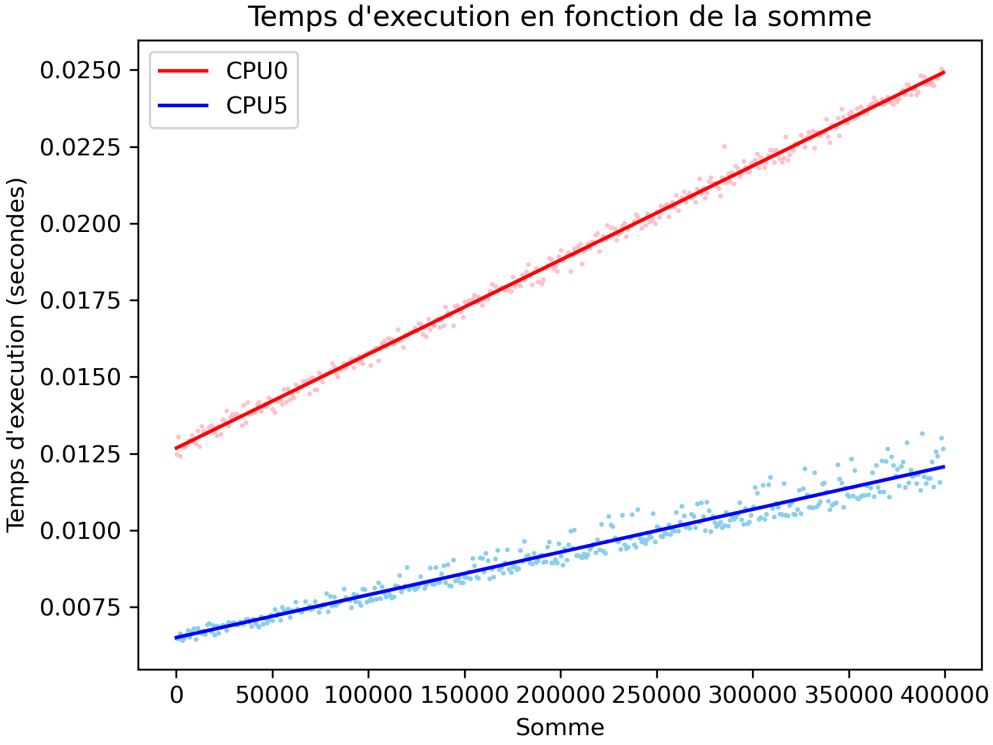

FIGURE 16 – Régression linéaire du temps d'exécution en fonction de l'entier sur lequel on réalise la somme

On obtient alors les régressions linéaires suivantes à l'aide d'un script `python` :

$$CPU0(n) = 3.065 \times 10^{-8} n + 0.0127$$

$$CPU5(n) = 1.393 \times 10^{-8} n + 0.0065$$

Les processeurs A53 sont à une fréquence de 1.5GHz, et les A72 à une fréquence de 2GHz. On peut donc voir que la pente de la régression linéaire est plus grande pour les A53 que pour les A72. Cela est cohérent avec le fait que les A53 sont moins puissants que les A72. Cependant, on ne retrouve pas le rapport de puissance entre les deux processeurs. En effet $\frac{3.065}{1.393} = 2.200 \neq \frac{2}{1.5} = 1.333$ ce qui signifie que la différence de performance entre les deux processeurs n'est pas uniquement due à la différence de fréquence.

Cela est dû au fait que les processeurs A72 ont d'autres avantages comme :

— Un parallélisme des instructions qu'il peut exécuter (*3 way super scalar*) contrairement aux 2 instructions en parallèle des A53

— *Out of order execution*, qui permet l'exécution dans le désordre de certaines instructions, ce qui permet de ne pas attendre qu'une instruction soit terminée pour en exécuter une autre

Ces deux avantages permettent aux A72 d'être plus performants que les A53, et donc d'avoir un temps d'exécution plus faible pour une même somme.

Je n'ai cependant pas eu le temps d'étudier cela plus en profondeur, par exemple, en mesurant le temps d'une manière à ne pas prendre en compte le démarrage des tâches.

5.2.3 Troisième idée : utilisation des instructions SIMD

Une dernière idée proposée par Antoine BERTOUT était l'utilisation des instructions SIMD. Ces instructions permettent de réaliser des opérations sur plusieurs entiers en même temps. Cela permet de réduire le temps d'exécution d'une opération. L'idée était alors d'utiliser ces instructions spécialisées, afin de mettre en valeur la différence de performance entre les deux processeurs. En effet, on s'attendait à ce que les A72 soient plus performants que les A53 sur ce type d'opération selon la documentation d'ARM.

On peut voir le code correspondant à ces essais dans la fonction `simd_test` du listing de code 15. Cependant, je n'ai pas pu générer des tâches suffisamment longues avec cette méthode, car je rencontrais un problème de mémoire avant d'avoir un temps d'exécution suffisamment long. Je n'ai pas pu identifier pourquoi on rencontrait ce problème si tôt, et je n'ai donc pas pu utiliser cette méthode pour générer des tâches avec des temps d'exécution différents.

6 Conclusion

Ce stage m'a permis de découvrir en profondeur le domaine de l'ordonnancement temps réel ainsi que le développement de module Linux. J'ai pu découvrir le monde de la recherche aux côtés de chercheur et doctorants qui m'ont permis de m'intégrer au sein du laboratoire.

J'ai pu grandement développer ma compréhension du noyau Linux et de l'implémentation d'ordonnanceurs sous un système d'exploitation.

L'objectif de mon stage était d'étudier la manière dont un algorithme d'ordonnancement pouvait être implémenté sur une plateforme hétérogène. J'ai pu étudier les différentes solutions possibles et les avantages et inconvénients de chacune. J'ai ensuite pu implémenter deux ordonnanceurs sur la plateforme qui m'a été fournie. J'ai pu étudier les résultats de ces implémentations et les comparer avec les résultats théoriques. J'ai aussi pu étudier les différentes solutions possibles pour générer des tâches qui mettraient en valeur la nature hétérogène de la plateforme, objectif qui n'avait pas été envisagé au début du stage.

Cependant je n'au pas pu implémenter un ordonnanceur qui utilise la nature hétérogène de la plateforme à son avantage. En effet, cela était un des objectifs finaux de mon stage, mais la complexité du noyau Linux, et du patch LITMUS^{RT}ne m'a pas permis de réaliser cela à temps. J'ai aussi passé une grande partie de mon temps à comprendre les mécanismes de migration de tâches et de job.

Pour conclure, mes travaux ont permis d'implémenter deux ordonnanceurs sur une plateforme hétérogène et d'étudier les résultats de ces implémentations. Cependant, je n'ai pas pu implémenter un ordonnanceur qui utilise la nature hétérogène de la plateforme à son avantage. Cependant, mon travail permettra aux chercheurs du projet SHRIMP d'implémenter sur la plateforme hétérogène les ordonnanceurs qu'ils concevront.

Références

[1] Antoine Bertout, Joël Goossens, Emmanuel Grolleau, and Xavier Poczekajlo. Workload assignment for global real-time scheduling on unrelated multicore platforms. In *Proceedings of the 28th International Conference on Real-Time Networks and Systems*, pages 139–148, 2020.

[2] B Brandenburg and J Anderson. Feather-trace : A lightweight event tracing toolkit. In *Proceedings of the third international workshop on operating systems platforms for embedded real-time applications*, pages 19–28. Citeseer, 2007.

[3] Bjorn B Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[4] Nick Desaulniers. Submitting your first patch to the linux kernel and responding to feedback. `https://nickdesaulniers.github.io/blog/2017/05/16/submitting-your-first-patch-to-the-linux-kernel-and-responding-to-feedback/`, 2021. Accessed : 2021-05-01.

Glossaire

bootloader court programme chargé au démarrage de l'ordinateur initialisant le système d'exploitation. 11

checksum md5 algorithme de hachage cryptographique de 128 bits permettant de produire un résultat (appelé aussi empreinte) à partir d'un fichier. 22

cluster ensemble interconnecté de plusieurs processeurs. 10

git système de gestion de versions décentralisé, utilisé pour suivre les modifications apportées à des fichiers sources dans un projet de développement logiciel. 11, 13

ordonnancement ou *scheduling* est l'activité qui consiste à affecter des ressources à des tâches. 6

ordonnanceur ou *scheduler* est un composant d'un système d'exploitation chargé de gérer l'ordonnancement des processus. 6, 16

plateforme hétérogène système formé d'un ensemble de processeurs différents. 6

préemption processus par lequel un système d'exploitation interrompt temporairement l'exécution d'une tâche en cours pour donner la priorité à une autre tâche de plus haute priorité. 17

processeur ou *CPU* est un composant présent dans tout ordinateur. Il est chargé d'effectuer les calculs et de gérer les flux de données. 17

SHRIMP *Scheduling of Real-Time Heterogeneous Multiprocessor Platform* ou Ordonnancement Temps réel de Plateforme Multiprocesseur Hétérogène. 6

SOC ou *Système On a Chip* est un circuit intégré qui rassemble sur une même puce plusieurs composants d'un ordinateur. 10

Table des figures

Listings

Annexe

```
#
# Makefile for LITMUS^RT
#

obj-y = sched_plugin.o litmus.o \
        preempt.o \
        litmus_proc.o \
        budget.o \
        clustered.o \
        jobs.o \
        sync.o \
        rt_domain.o \
        edf_common.o \
        fp_common.o \
        fdso.o \
        locking.o \
        srp.o \
        bheap.o \
        binheap.o \
        ctrldev.o \
        uncachedev.o \
        sched_gsn_edf.o \
        sched_psn_edf.o \
        sched_pfp.o \
        sched_p_edf.o

obj-$(CONFIG_PLUGIN_CEDF) += sched_cedf.o
obj-$(CONFIG_PLUGIN_PFAIR) += sched_pfair.o

obj-$(CONFIG_FEATHER_TRACE) += ft_event.o ftdev.o
obj-$(CONFIG_SCHED_TASK_TRACE) += sched_task_trace.o
obj-$(CONFIG_SCHED_DEBUG_TRACE) += sched_trace.o
obj-$(CONFIG_SCHED_OVERHEAD_TRACE) += trace.o

obj-y += sched_pres.o

obj-y += reservations/
```

Listing 9 – linux/litmus/Makefile

```
#include <linux/module.h>
#include <linux/percpu.h>
#include <linux/sched.h>
#include <litmus/litmus.h>
#include <litmus/budget.h>
#include <litmus/edf_common.h>
#include <litmus/jobs.h>
#include <litmus/litmus_proc.h>
#include <litmus/debug_trace.h>
#include <litmus/preempt.h>
#include <litmus/rt_domain.h>
#include <litmus/sched_plugin.h>
#include <litmus/sched_trace.h>

struct p_edf_cpu_state {
	rt_domain_t local_queues;
	int cpu;
	struct task_struct* scheduled;
};

static DEFINE_PER_CPU(struct p_edf_cpu_state, p_edf_cpu_state);

#define cpu_state_for(cpu_id) (&per_cpu(p_edf_cpu_state, cpu_id))
#define local_cpu_state()    (this_cpu_ptr(&p_edf_cpu_state))
#define remote_edf(cpu)  (&per_cpu(p_edf_cpu_state, cpu).local_queues)
#define remote_pedf(cpu) (&per_cpu(p_edf_cpu_state, cpu))
#define task_edf(task)   remote_edf(get_partition(task))

static struct domain_proc_info p_edf_domain_proc_info;

static long p_edf_get_domain_proc_info(struct domain_proc_info **ret)
{
	*ret = &p_edf_domain_proc_info;
	return 0;
}

static void p_edf_setup_domain_proc(void)
{
	int i, cpu;
	int num_rt_cpus = num_online_cpus();

	struct cd_mapping *cpu_map, *domain_map;

	memset(&p_edf_domain_proc_info, 0, sizeof(p_edf_domain_proc_info));
	init_domain_proc_info(&p_edf_domain_proc_info, num_rt_cpus, num_rt_cpus);
	p_edf_domain_proc_info.num_cpus = num_rt_cpus;
	p_edf_domain_proc_info.num_domains = num_rt_cpus;

	i = 0;
	for_each_online_cpu(cpu) {
		cpu_map = &p_edf_domain_proc_info.cpu_to_domains[i];
		domain_map = &p_edf_domain_proc_info.domain_to_cpus[i];

		cpu_map->id = cpu;
		domain_map->id = i;
		cpumask_set_cpu(i, cpu_map->mask);
		cpumask_set_cpu(cpu, domain_map->mask);
		++i;
	}
}
```

```
61
62  /* This helper is called when task ‘prev‘ exhausted its budget or when
63  * it signaled a job completion. */
64  static void p_edf_job_completion(struct task_struct *prev, int budget_exhausted)
65  {
66          sched_trace_task_completion(prev, budget_exhausted);
67      TRACE_TASK(prev, "job_completion(forced=%d).\n", budget_exhausted);
68
69      tsk_rt(prev)->completed = 0;
70          /* Call common helper code to compute the next release time, deadline,
71          * etc. */
72          prepare_for_next_period(prev);
73  }
74
75  /* Add the task ‘tsk‘ to the appropriate queue. Assumes the caller holds the
        ready lock.
76  */
77  static void p_edf_requeue(struct task_struct *tsk, struct p_edf_cpu_state *
        cpu_state)
78  {
79          if (is_released(tsk, litmus_clock())) {
80                  /* Uses __add_ready() instead of add_ready() because we already
81                      * hold the ready lock. */
82                  __add_ready(&cpu_state->local_queues, tsk);
83                  TRACE_TASK(tsk, "added to ready queue on reschedule\n");
84          } else {
85                  /* Uses add_release() because we DON'T have the release lock. */
86                  add_release(&cpu_state->local_queues, tsk);
87                  TRACE_TASK(tsk, "added to release queue on reschedule\n");
88          }
89  }
90
91  static int p_edf_check_for_preemption_on_release(rt_domain_t *local_queues)
92  {
93          struct p_edf_cpu_state *state = container_of(local_queues,
94                                                       struct p_edf_cpu_state,
95                                                           local_queues);
96
97          /* Because this is a callback from rt_domain_t we already hold
98              * the necessary lock for the ready queue. */
99
100         if (edf_preemption_needed(local_queues, state->scheduled)) {
101                 preempt_if_preemptable(state->scheduled, state->cpu);
102                 return 1;
103         }
104         return 0;
105 }
106
107 static long p_edf_activate_plugin(void)
108 {
109         int cpu;
110         struct p_edf_cpu_state *state;
111         for_each_online_cpu(cpu) {
112                 TRACE("Initializing CPU%d...\n", cpu);
113                 state = cpu_state_for(cpu);
114                 state->cpu = cpu;
115                 state->scheduled = NULL;
116                 edf_domain_init(&state->local_queues,
117                                 p_edf_check_for_preemption_on_release,
118                                 NULL);
119         }
```

```
120
121         p_edf_setup_domain_proc();
122         return 0;
123 }
124
125 static long p_edf_deactivate_plugin(void)
126 {
127         destroy_domain_proc_info(&p_edf_domain_proc_info);
128         return 0;
129 }
130
131
132
133 static struct task_struct* p_edf_schedule(struct task_struct * prev)
134 {
135         struct p_edf_cpu_state *local_state = local_cpu_state();
136
137         /* next == NULL means "schedule background work". */
138         struct task_struct *next = NULL;
139
140         /* prev's task state */
141         int exists, out_of_time, job_completed, self_suspends, preempt, resched;
142
143         raw_spin_lock(&local_state->local_queues.ready_lock);
144
145         BUG_ON(local_state->scheduled && local_state->scheduled != prev);
146         BUG_ON(local_state->scheduled && !is_realtime(prev));
147
148         exists = local_state->scheduled != NULL;
149         self_suspends = exists && !is_current_running();
150         out_of_time = exists && budget_enforced(prev) && budget_exhausted(prev);
151         job_completed = exists && is_completed(prev);
152
153         /* preempt is true if task `prev` has lower priority than something on
154         * the ready queue. */
155         preempt = edf_preemption_needed(&local_state->local_queues, prev);
156
157         /* check all conditions that make us reschedule */
158         resched = preempt;
159
160         /* if `prev` suspends, it CANNOT be scheduled anymore => reschedule */
161         if (self_suspends) {
162                 resched = 1;
163         }
164
165         /* also check for (in-)voluntary job completions */
166         if (out_of_time || job_completed) {
167                 p_edf_job_completion(prev, out_of_time);
168                 resched = 1;
169         }
170
171         if (resched) {
172                 /* First check if the previous task goes back onto the ready
173                 * queue, which it does if it did not self_suspend.
174                 */
175                 if (exists && !self_suspends) {
176                         p_edf_requeue(prev, local_state);
177                 }
178                 next = __take_ready(&local_state->local_queues);
179         } else {
180                 /* No preemption is required. */
```

```
181                 next = local_state->scheduled;
182         }
183
184         local_state->scheduled = next;
185         if (exists && prev != next) {
186                 TRACE_TASK(prev, "descheduled.\n");
187         }
188         if (next) {
189                 TRACE_TASK(next, "scheduled.\n");
190         }
191
192         /* This mandatory. It triggers a transition in the LITMUS^RT remote
193         * preemption state machine. Call this AFTER the plugin has made a
194         * local scheduling decision.
195         */
196         sched_state_task_picked();
197
198         raw_spin_unlock(&local_state->local_queues.ready_lock);
199         return next;
200 }
201
202 static long p_edf_admit_task(struct task_struct *tsk)
203 {
204         if (task_cpu(tsk) == get_partition(tsk)) {
205                 TRACE_TASK(tsk, "accepted by p_edf plugin.\n");
206                 return 0;
207         }
208         return -EINVAL;
209 }
210
211 static void p_edf_task_new(struct task_struct *tsk, int on_runqueue,
212                            int is_running)
213 {
214         /* We'll use this to store IRQ flags. */
215         unsigned long flags;
216         struct p_edf_cpu_state *state = cpu_state_for(get_partition(tsk));
217         lt_t now;
218
219         TRACE_TASK(tsk, "is a new RT task %llu (on runqueue:%d, running:%d)\n",
220                    litmus_clock(), on_runqueue, is_running);
221
222         /* Acquire the lock protecting the state and disable interrupts. */
223         raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);
224
225         now = litmus_clock();
226
227         /* Release the first job now. */
228         release_at(tsk, now);
229
230         if (is_running) {
231                 /* If tsk is running, then no other task can be running
232                     * on the local CPU. */
233                 BUG_ON(state->scheduled != NULL);
234                 state->scheduled = tsk;
235         } else if (on_runqueue) {
236                 p_edf_requeue(tsk, state);
237         }
238
239         if (edf_preemption_needed(&state->local_queues, state->scheduled))
240                 preempt_if_preemptable(state->scheduled, state->cpu);
241
```

```
242         raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
243 }
244
245 static void p_edf_task_exit(struct task_struct *tsk)
246 {
247         unsigned long flags;
248         struct p_edf_cpu_state *state = cpu_state_for(get_partition(tsk));
249         raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);
250         rt_domain_t*      edf;
251
252         /* For simplicity, we assume here that the task is no longer queued
                anywhere else. This
253             * is the case when tasks exit by themselves; additional queue
                    management is
254             * is required if tasks are forced out of real-time mode by other tasks
                    . */
255
256         if (is_queued(tsk)){
257                 edf = task_edf(tsk);
258                 remove(edf,tsk);
259         }
260
261         if (state->scheduled == tsk) {
262                 state->scheduled = NULL;
263         }
264
265         preempt_if_preemptable(state->scheduled, state->cpu);
266         raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
267 }
268
269 /* Called when the state of tsk changes back to TASK_RUNNING.
270     * We need to requeue the task.
271     *
272     * NOTE: If a sporadic task is suspended for a long time,
273     * this might actually be an event-driven release of a new job.
274     */
275 static void p_edf_task_resume(struct task_struct *tsk)
276 {
277         unsigned long flags;
278         struct p_edf_cpu_state *state = cpu_state_for(get_partition(tsk));
279         lt_t now;
280         TRACE_TASK(tsk, "wake_up at %llu\n", litmus_clock());
281         raw_spin_lock_irqsave(&state->local_queues.ready_lock, flags);
282
283         now = litmus_clock();
284
285         if (is_sporadic(tsk) && is_tardy(tsk, now)) {
286                 /* This sporadic task was gone for a "long" time and woke up past
287                     * its deadline. Give it a new budget by triggering a job
288                     * release. */
289                 inferred_sporadic_job_release_at(tsk, now);
290                 TRACE_TASK(tsk, "woke up too late.\n");
291         }
292
293         /* This check is required to avoid races with tasks that resume before
294             * the scheduler "noticed" that it resumed. That is, the wake up may
295             * race with the call to schedule(). */
296         if (state->scheduled != tsk) {
297                 TRACE_TASK(tsk, "is being reqeued\n");
298                 p_edf_requeue(tsk, state);
299                 if (edf_preemption_needed(&state->local_queues, state->scheduled))
```

```
                    {
300                   preempt_if_preemptable(state->scheduled, state->cpu);
301             }
302     }
303
304     raw_spin_unlock_irqrestore(&state->local_queues.ready_lock, flags);
305 }
306
307
308 static struct sched_plugin p_edf_plugin = {
309     .plugin_name          = "P-EDF",
310     .schedule             = p_edf_schedule,
311     .task_wake_up         = p_edf_task_resume,
312     .admit_task           = p_edf_admit_task,
313     .task_new             = p_edf_task_new,
314     .task_exit            = p_edf_task_exit,
315     .get_domain_proc_info = p_edf_get_domain_proc_info,
316     .activate_plugin      = p_edf_activate_plugin,
317     .deactivate_plugin    = p_edf_deactivate_plugin,
318     .complete_job         = complete_job,
319 };
320
321 static int __init init_p_edf(void)
322 {
323     return register_sched_plugin(&p_edf_plugin);
324 }
325
326 module_init(init_p_edf);
```

Listing 10 – linux/litmus/sched_p_edf.c

```
# LITMUS^RT
#

#
# Scheduling
#
CONFIG_PLUGIN_PFAIR=y
# CONFIG_RELEASE_MASTER is not set
CONFIG_PREFER_LOCAL_LINKING=y
CONFIG_LITMUS_QUANTUM_LENGTH_US=1000
CONFIG_BUG_ON_MIGRATION_DEADLOCK=y
# end of Scheduling

#
# Real-Time Synchronization
#
CONFIG_NP_SECTION=y
CONFIG_LITMUS_LOCKING=y
# end of Real-Time Synchronization

#
# Performance Enhancements
#
CONFIG_ALLOW_EARLY_RELEASE=y
# CONFIG_EDF_TIE_BREAK_LATENESS is not set
CONFIG_EDF_TIE_BREAK_LATENESS_NORM=y
# CONFIG_EDF_TIE_BREAK_HASH is not set
# CONFIG_EDF_PID_TIE_BREAK is not set
# end of Performance Enhancements

#
# Tracing
#
CONFIG_FEATHER_TRACE=y
CONFIG_SCHED_TASK_TRACE=y
CONFIG_SCHED_TASK_TRACE_SHIFT=9
CONFIG_SCHED_OVERHEAD_TRACE=y
CONFIG_SCHED_OVERHEAD_TRACE_SHIFT=22
CONFIG_SCHED_DEBUG_TRACE=y
CONFIG_SCHED_DEBUG_TRACE_SHIFT=18
CONFIG_SCHED_DEBUG_TRACE_CALLER=y
# CONFIG_PREEMPT_STATE_TRACE is not set
# CONFIG_REPORT_TIMER_LATENCY is not set
# end of Tracing
# end of LITMUS^RT
```

Listing 11 – Partie du fichier .config liée a LITMUS^{RT}

```
cpus {
        #address-cells = <2>;
        #size-cells = <0>;

        cpu-map {
                cluster0 {
                        core0 {
                                cpu = <&cpu_l0>;
                        };
                        core1 {
                                cpu = <&cpu_l1>;
                        };
                        core2 {
                                cpu = <&cpu_l2>;
                        };
                        core3 {
                                cpu = <&cpu_l3>;
                        };
                };

                cluster1 {
                        core0 {
                                cpu = <&cpu_b0>;
                        };
                        core1 {
                                cpu = <&cpu_b1>;
                        };
                };
        };

        cpu_l0: cpu@0 {
                device_type = "cpu";
                compatible = "arm,cortex-a53";
                reg = <0x0 0x0>;
                enable-method = "psci";
                next-level-cache = <&l2_0>;
                capacity-dmips-mhz = <485>;
                clocks = <&cru ARMCLKL>;
                #cooling-cells = <2>; /* min followed by max */
                dynamic-power-coefficient = <100>;
                cpu-idle-states = <&CPU_SLEEP &CLUSTER_SLEEP>;

                l2_0: l2-cache {
                compatible = "cache,arm,arch-cache";
        };
        };

        cpu_l1: cpu@1 {
                device_type = "cpu";
                compatible = "arm,cortex-a53";
                reg = <0x0 0x1>;
                enable-method = "psci";
                next-level-cache = <&l2_0>;
                capacity-dmips-mhz = <485>;
                clocks = <&cru ARMCLKL>;
                #cooling-cells = <2>; /* min followed by max */
                dynamic-power-coefficient = <100>;
                cpu-idle-states = <&CPU_SLEEP &CLUSTER_SLEEP>;
        };

```

```
61          cpu_l2: cpu@2 {
62                  device_type = "cpu";
63                  compatible = "arm,cortex-a53";
64                  reg = <0x0 0x2>;
65                  enable-method = "psci";
66                  next-level-cache = <&l2_0>;
67                  capacity-dmips-mhz = <485>;
68                  clocks = <&cru ARMCLKL>;
69                  #cooling-cells = <2>; /* min followed by max */
70                  dynamic-power-coefficient = <100>;
71                  cpu-idle-states = <&CPU_SLEEP &CLUSTER_SLEEP>;
72          };
73
74          cpu_l3: cpu@3 {
75                  device_type = "cpu";
76                  compatible = "arm,cortex-a53";
77                  reg = <0x0 0x3>;
78                  enable-method = "psci";
79                  next-level-cache = <&l2_0>;
80                  capacity-dmips-mhz = <485>;
81                  clocks = <&cru ARMCLKL>;
82                  #cooling-cells = <2>; /* min followed by max */
83                  dynamic-power-coefficient = <100>;
84                  cpu-idle-states = <&CPU_SLEEP &CLUSTER_SLEEP>;
85          };
86
87          cpu_b0: cpu@100 {
88                  device_type = "cpu";
89                  compatible = "arm,cortex-a72";
90                  reg = <0x0 0x100>;
91                  enable-method = "psci";
92                  next-level-cache = <&l2_1>;
93                  capacity-dmips-mhz = <1024>;
94                  clocks = <&cru ARMCLKB>;
95                  #cooling-cells = <2>; /* min followed by max */
96                  dynamic-power-coefficient = <436>;
97                  cpu-idle-states = <&CPU_SLEEP &CLUSTER_SLEEP>;
98
99                  l2_1: l2-cache {
100                 compatible = "cache,arm,arch-cache";
101         };
102         };
103
104         cpu_b1: cpu@101 {
105                 device_type = "cpu";
106                 compatible = "arm,cortex-a72";
107                 reg = <0x0 0x101>;
108                 enable-method = "psci";
109                 next-level-cache = <&l2_1>;
110                 capacity-dmips-mhz = <1024>;
111                 clocks = <&cru ARMCLKB>;
112                 #cooling-cells = <2>; /* min followed by max */
113                 dynamic-power-coefficient = <436>;
114                 cpu-idle-states = <&CPU_SLEEP &CLUSTER_SLEEP>;
115         };
116         ...
117 }
```

Listing 12 – Modifications apportées au fichier `rk3399.dtsi`

```c
/*
* litmus/rm_common.c
*/
#include <linux/percpu.h>
#include <linux/sched.h>
#include <linux/list.h>

#include <litmus/litmus.h>
#include <litmus/sched_plugin.h>
#include <litmus/sched_trace.h>
#include <litmus/debug_trace.h>

#include <litmus/rm_common.h>


/* rm_higher_prio - returns true if first has a higher RM priority
*                  than second. Deadline ties are broken by PID.
*
* both first and second may be NULL
*/
int rm_higher_prio(struct task_struct* first,
                   struct task_struct* second)
{
        struct task_struct *first_task = first;
        struct task_struct *second_task = second;

        /* There is no point in comparing a task to itself. */
        if (first && first == second) {
                TRACE_TASK(first,
                           "WARNING: pointless edf priority comparison.\n");
                return 0;
        }


        /* check for NULL tasks */
        if (!first || !second)
                return first && !second;


        if (shorter_exec_time(first_task, second_task)) {
                return 1;
        }
        else if (get_rt_period(first_task) == get_rt_period(second_task)) {
                /* Need to tie break. All methods must set pid_break to 0/1 if
                * first_task does not have priority over second_task.
                */
                int pid_break;

                /* CONFIG_EDF_PID_TIE_BREAK */
                pid_break = 1; // fall through to tie-break by pid;


                /* Tie break by pid */
                if(pid_break) {
                        if (first_task->pid < second_task->pid) {
                                return 1;
                        }
                        else if (first_task->pid == second_task->pid) {
                                /* If the PIDs are the same then the task with the
                                * inherited priority wins.
```

```
61                              */
62                              if (!second->rt_param.inh_task) {
63                                      return 1;
64                              }
65                      }
66              }
67      }
68      return 0; /* fall-through. prio(second_task) > prio(first_task) */
69  }
70
71  int rm_ready_order(struct bheap_node* a, struct bheap_node* b)
72  {
73      return rm_higher_prio(bheap2task(a), bheap2task(b));
74  }
75
76  void rm_domain_init(rt_domain_t* rt, check_resched_needed_t resched,
77                                      release_jobs_t release)
78  {
79      rt_domain_init(rt, rm_ready_order, resched, release);
80  }
81
82
83  /* need_to_preempt - check whether the task t needs to be preempted
84  *                  call only with irqs disabled and with ready_lock acquired
85  *                  THIS DOES NOT TAKE NON-PREEMPTIVE SECTIONS INTO ACCOUNT!
86  */
87  int rm_preemption_needed(rt_domain_t* rt, struct task_struct *t)
88  {
89      /* we need the read lock for edf_ready_queue */
90      /* no need to preempt if there is nothing pending */
91      if (!__jobs_pending(rt))
92              return 0;
93      /* we need to reschedule if t doesn't exist */
94      if (!t)
95              return 1;
96
97      /* NOTE: We cannot check for non-preemptibility since we
98      *       don't know what address space we're currently in.
99      */
100
101     /* make sure to get non-rt stuff out of the way */
102     return !is_realtime(t) || rm_higher_prio(__next_ready(rt), t);
103 }
```

Listing 13 – litmus/rm_common.c

```
#!/bin/bash
start="$(date +'%s.%N')"
$@
end="$(date +'%s.%N')"
diff=$(awk "BEGIN { print $end - $start }")
echo "$diff" >&2
```

Listing 14 – Script de mesure du temps d'éxécution

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <arm_neon.h>


int simd_test(int array_size){
    float32x4_t vec = vdupq_n_f32(0.1f);

    float array[array_size];
    float result[array_size];
    float sum = 0.0f;

    for (int i = 0; i < array_size; i++) {
        array[i] = (float)i;
    }

    for (int i = 0; i < array_size; i += 4) {
        if (i + 3 < array_size) { // Check bounds to avoid going over the end eof the array
            float32x4_t data = vld1q_f32(&array[i]);
            float32x4_t result_vec = vmulq_f32(data, vec);
            vst1q_f32(&result[i], result_vec);
        }
    }

    for (int i = 0; i < array_size; i++) {
        sum += result[i];
    }

    printf("Sum: %f\n", sum);

    return 0;
}

int for_loop_sum(int range)
{
    uint64_t k;
    volatile uint64_t c = 0;
    for(k = 0; k < range; k++){
        c += k;
    }
    return 0;
}




int main(int argc, char *argv[]) {

    int argument;

    if (argc < 2){
        printf("Usage : %s [options] \n", argv[0]);
        return 1;
    }


    for(int i = 1; i < argc; i++){
        if(strcmp(argv[i], "-h") == 0 || strcmp(argv[i], "---help") == 0){
```

```
60              //Display of help message
61              printf("Usage : %s [options]\n", argv[0]);
62              printf("Options : \n");
63              printf("-h or --help Display this message\n");
64              printf("--simd [arraySize] runs multiple simd operations\n");
65              printf("--sum [range] runs a sum up to `range`\n");
66          } else if (strcmp(argv[i], "--simd") == 0){
67              if (i+1<argc){
68                  argument = atoi(argv[i+1]);
69                  printf("running simd_test with arraysize = %i\n", argument);
70                  return simd_test(argument);
71              } else {
72                  printf("--simd requires an array size on which the operations will
                        be performed\n");
73                  return 1; //early return, incorect arguments
74              }
75          }
76          else if (strcmp(argv[i], "--sum") == 0){
77              if (i+1<argc){
78                  argument = atoi(argv[i+1]);
79                  printf("running for_loop_sum with range = %i\n", argument);
80                  return for_loop_sum(argument);
81              } else {
82                  printf("--sum requires an range up to which the operations will be
                        performed\n");
83                  return 1; //early return, incorect arguments
84              }
85          }
86      }
87  }
```

Listing 15 – exec-time-tester.c