Titre: Title:	Analyse de faisabilité de l'implantation d'un protocole de communication sur processeur multicoeurs
Auteur: Author:	Michel Gémieux
Date:	2015
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Gémieux, M. (2015). Analyse de faisabilité de l'implantation d'un protocole de communication sur processeur multicoeurs [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1709/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/1709/
Directeurs de recherche: Advisors:	Yvon Savaria, & Guchuan Zhu
Programme: Program:	génie électrique

UNIVERSITÉ DE MONTRÉAL

ANALYSE DE FAISABILITÉ DE L'IMPLANTATION D'UN PROTOCOLE DE COMMUNICATION SUR PROCESSEUR MULTICOEURS

MICHEL GÉMIEUX

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE ÉLECTRIQUE)

AVRIL 2015

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ANALYSE DE FAISABILITÉ DE L'IMPLANTATION D'UN PROTOCOLE DE COMMUNICATION SUR PROCESSEUR MULTICOEURS

présenté par : GÉMIEUX Michel

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

M. DAVID Jean-Pierre, Ph. D., président

M. SAVARIA Yvon, Ph. D., membre et directeur de recherche

M. ZHU Guchuan, Doctorat, membre et codirecteur de recherche

M. CARDINAL Christian, Ph. D., membre

DÉDICACE

À Camille, pour tout son support tout au long de ce grand périple.

REMERCIEMENTS

J’aimerais commencer par remercier Yvon Savaria et Guchuan Zhu qui m’ont accepté sous leur direction pour mener ce projet de maitrise au but. Je leur serais immuablement reconnaissant pour m’avoir permis d’adhérer à un projet sous la cotutelle d’une entreprise travaillant sur des projets à la pointe de la technologie. Un projet qui m’a fait acquérir une meilleure autonomie, mais surtout des compétences pertinentes au gout du jour au sein du marché de la microélectronique.

Je voudrai aussi remercier Patrice Plante et Sébastien Regimbal qui m’ont guidé et aidé de l’introduction du projet jusqu’à la finalisation de ce mémoire.

Un merci particulier à Christian Cardinal et Jean-Pierre David d’avoir accepté de donner de leurs temps afin de faire partie du jury qui évaluera le travail effectué tout au long de ma maitrise.

En dernier, mais non les moindres je tiens à remercier les personnes proches de moi, que ce soit du domaine personnel que professionnel, m’ayant soutenue à travers ce merveilleux, mais aussi tumultueux périple. Particulièrement à :

- Mes camarades de salle de laboratoire : Gontran Sion, Sylvain Charasse, Thalie Keklikian, Matthieu Courbariaux et Safa Berima.
- L’ensemble de l’administration du GRM et de Polytechnique, toujours prêt à aider les étudiants au meilleur de leurs habilités.
- L’ensemble technique du GRM, je pense surtout à Rejean Lepage et Jean Bouchard qui ont toujours été disponible à aider, quel que soit le problème, lors de nos travaux et expérimentations.
- Ma famille proche et lointaine, pour leur soutien incessant.

RÉSUMÉ

Les travaux de ce mémoire s'inscrivent dans le cadre d'un projet qui fait l'objet d'un parrainage industriel. Les résultats visent à comprendre le comportement d'un système de traitement opérant dans des contextes précis. Nous situons ce projet à l'intersection des principes d'ordonnancements de tâches, des systèmes d'exécution, de la virtualisation de fonctions de réseaux et surtout les contraintes associées à la virtualisation d'une pile de protocole LTE (Long Term Evolution), la norme de téléphonie cellulaire la plus en vue en ce moment. Une revue de littérature est proposée pour expliquer en détail les concepts vus plus haut, afin d'avoir une idée précise de la situation de test.

D'abord, une étude des grappes d'unités de traitement temps réel est effectuée dans l'optique de l'implémentation de ce qu'il est convenu d'appeler un Cloud Radio Area Network (C-RAN), qui supporte sur une plateforme infonuagique l'électronique qui effectue le traitement de signal requis pour un point d'accès de téléphonie cellulaire. L'étude développée dans ce mémoire vise à évaluer les différents goulots d'étranglement qui peuvent survenir suite à la réception d'un paquet LTE au sein d'une trame CPRI (Common Public Radio Interface), jusqu'à l'envoi de ce paquet d'un serveur maitre jusqu'aux esclaves. Nous évaluons donc les latences et bandes passantes observées pour les différents protocoles composant la plateforme. Nous caractérisons notamment les communications CPRI des antennes vers le bassin de stations de base virtuelles, une communication de type Quick Path Interconnect (QPI) entre des cœurs de traitement et un réseau logique programmable de type FPGA, une communication dédiée point à point entre le FPGA et une carte NIC (Network Interface Card) pour finir avec l'envoi de trames Ethernet vers les serveurs esclaves. Cette étude nous permet de déduire que la virtualisation d'une pile LTE est viable sur une telle grappe de calcul temps réel.

Ensuite, pour que l'on puisse valider l'efficacité de différents algorithmes d'ordonnancement, une émulation de virtualisation d'un Uplink LTE sera faite. Par le biais d'un système d'exécution nommé StarPU couplé avec des outils de profilage, nous obtenons des résultats permettant d'évaluer la nécessité d'unité d'exécution dédiées pour la gestion de tâches au sein d'un serveur.

Enfin, nous démontrons que la gestion et l'exécution des tâches associées à la virtualisation d'une pile LTE est faisable de manière logicielle avec deux des algorithmes d'ordonnancement testés sur six. De plus, nous ajoutons le concept d'ordonnancement partagé à travers de multiples unités de calcul, qui s'avèrent plus efficace que l'implémentation d'un cœur de processeur dédié à la gestion de tâches.

ABSTRACT

The work performed as part of this Master thesis is done in the context of an industrially sponsored project. The objective is to understand the runtime behavior of a class of systems in specific contexts. We place this project at the intersection of the principles of task scheduling, runtimes, Network Functions Virtualisation (NFVs) and especially with the constraints associated with virtualization of an LTE (Long Term Evolution) stack that is the most prominent cellular telecommunication standard at the moment. A literature review is proposed to explain in detail the concepts discussed above, in order to have a clear idea of the target environment.

First, a study of a real time processing cluster is carried out in relation to the implementation of the so-called Cloud Radio Area Network (C-RAN) that supports on a cloud platform all the electronics which performs the signal processing required for a cellular access point. The study developed in this paper is to evaluate the various bottlenecks that can occur following the receipt of an LTE packet within a Common Public Radio Interface (CPRI) frame, then as part of sending the package to a master server before routing it to the slaves. We evaluate the latencies and bandwidths observed for the different protocols used on the platform components. In particular, we characterize the CPRI communications from the antennas to the virtual base stations units, a Quick Path Interconnect (QPI) communication between processing cores and a programmable logic array in the type of a FPGA, a dedicated point to point communication between the FPGA and a NIC (Network Interface Card) to end with the sending Ethernet frames to the slave servers. This study allows us to infer that the virtualization of an LTE stack is viable on a real time computation cluster with the implied architecture.

Then, to be able to validate the effectiveness of different scheduling algorithms, an emulation of a LTE Uplink stack virtualization will be made. Through a runtime called StarPU coupled with profiling tools, we deliver results to assess the need for dedicated thread or cores to manage tasks within a server.

Finally, we demonstrate that the management and execution of the tasks associated with the virtualization of a LTE stack is feasible in software with two of the scheduling algorithms taken from a set of six that were tested. In addition, we add a scheduling concept shared across multiple

computing units, which is more efficient than the implementation of a dedicated processor to manage tasks.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

ACK	Acknowledge
ASIC	Application Specific Integrated Circuit
BBU	Base Band Unit
BRAM	Bloc RAM
BS	Base Station
CAPEX	Capital Expenses
CAPI	Coherent Accelerator Processor Interface
CCI	Cache Coherent Interface
CPRI	Common Public Radio Interface
CPU	Central Processing Unit
CRAN	Cloud/Central Radio Area Network
DDR	Double Data Rate
DPDK	Data Plane Development Kit
eNodeB	Evolved NodeB
FDMA	Frequency Division Multiple Access
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
GDDR	Graphics Double Data Rate
GPP	General Purpose Processor
GSM	Global System/Standard Mobile
HPC	High Performance Computing
IP	Intellectual Property

IQ	In-phase and Quadrature
LLC	Last Level of Cache
LTE	Long Term Evolution
MAC	Media Access Control
MIC	Many Integrated Core
MIMO	Multiple In Multiple Out
MKL	Math Kernel Library
NFV	Network Function Virtualization
NIC	Network Interface Controller
OBSAI	Open Base Station Architecture Initiative
OFDM	Orthogonal Frequency-Division Multiplexing
OPEX	Operation Expenses
PCIe	Peripheral Component Interconnect Express
PHY	Physical Layer
QPI	Quick Path Interconnect
RAN	Radio Area Network
RRH	Remote Radio Head
RRU	Remote Radio Unit
SDN	Software Defined Network
SIMD	Single Instruction on Multiple Data
TBB	Threading Building Blocks
WAN	Wide Area Network
WCET	Worst Case Execution Tim

CHAPITRE 1 INTRODUCTION

En 2015, nous vivons dans un environnement centré sur l'accessibilité de l'information. Nous sommes connectés en permanence et voulons que nos moindres faits et gestes soient répertoriés, traités, analysés pour se retrouver dans le « nuage » [1]. Nous sommes dans le monde du *« Big Data*». Ainsi, ce projet se situe donc dans le monde infonuagique, plus précisément au niveau des centres de données. Nous cherchons à améliorer les traitements de tâches dans un contexte d'opération *Cloud Radio Access Network* (C-RAN), axé sur la norme *Long Term Evolution* (LTE). Pour ce faire nous voulons exploiter les bienfaits d'un système double socle hétérogène, où le premier est habité par un processeur multi-cœurs d'Intel et l'autre par un *Field Programmable Gate Array* (FPGA) de Xilinx. Le FPGA servira de plateforme parallèle comportant des accélérateurs et gestionnaires de tâches matérielles.

Une meilleure gestion des tâches ainsi que l'accélération des applications roulant sur de multiples serveurs [2] (contexte C-RAN), nous permettra une utilisation plus efficace et adéquate d'une grappe de calcul. Cette recherche vise à déployer des systèmes qui consomment substantiellement moins d'énergie et pour lesquels les coûts d'installations sont réduits, ce qui peut en réduire l'impact négatif sur la planète (l'empreinte de carbone).

Dans cette introduction nous allons tout abord définir les concepts de base associés au projet de recherche documenté par ce mémoire. Ensuite, nous situerons les travaux effectués en exposant la problématique étudiée. Puis seront listés les différents objectifs de recherche suivis au long de cette maîtrise. Enfin, l'organisation des différents chapitres et du contenu de ce mémoire sera détaillée.

1.1 Définitions et concepts de base

Commençons par le cœur de la plateforme, l'unité de calcul, plus communément appelée le microprocesseur. Il s'agit d'un circuit intégré qui a pour but d'exécuter des instructions et de traiter des données. Son comportement est généralement séquentiel, c'est-à-dire qu'il effectue une tâche à la fois. Afin de pouvoir traiter les nouveaux programmes de plus en plus demandant en puissance de calcul, la fréquence des processeurs ne cessa d'augmenter depuis les années 70 jusqu'à assez récemment. Cependant, dû à la stagnation de la fréquence d'opération des processeurs depuis

quelques années en raison des contraintes de puissance et du problème de courant de fuites, les fournisseurs de matériel informatique ont cherché à augmenter le parallélisme dans leurs processeurs afin d'améliorer les performances des applications scientifiques et d'ingénierie hautement parallèles. Cela a conduit à l'arrivée des puces dites « *many cores* » [3], qui sont des systèmes comportant un nombre élevé (plus de 60) de cœurs de calcul souvent cadencés à une fréquence d'horloge un peu moins élevée (1 GHz) que les processeurs standards, ainsi qu'à l'introduction de processeurs multi cœurs [3] qui ont un nombre plus restreint d'unités de calcul (8 par exemple), mais qui sont cadencés à une fréquence plus élevée (3 GHz).

Ce mémoire explore comment exécuter des fonctions complexes sur ces fameuses unités de calcul. C'est ici qu'entrent en jeux, les systèmes d'exécution [4]. Ces systèmes fournissent une couche logicielle associée au système d'exploitation qui permet de gérer l'exécution de fonctions sur les cœurs des microprocesseurs. Il existe plusieurs variantes à travers la littérature, toutes avec leurs particularités. La performance d'un système d'exécution dépend du profil des fonctions à exécuter, elles peuvent être sur la forme de tâche, ce qu'on appelle la programmation basée sur les tâches ou être sous la forme de fils [5] (*Pthreads*).

Puis, au sein du système d'exécution, on retrouve l'ordonnanceur et son algorithme d'ordonnancement. L'ordonnanceur est le mécanisme qui s'occupe de gérer quel processus peut avoir accès aux ressources de la plateforme et quand il peut le faire. Dans notre cas nous parlerons d'un ordonnanceur de tâches [6]. Afin que ce mécanisme soit efficace, il est jumelé à un algorithme. Le choix de l'algorithme est entièrement dépendant du contexte et du type de tâches à traiter. Il existe deux grandes familles d'algorithmes d'ordonnancement, les dynamiques et les statiques [7]. Chacune de ces familles a des avantages et inconvénients propres.

Cela nous mène au contexte dans lequel nous voulons faire un ordonnancement de tâches sur multiprocesseur. Le *Cloud Radio Access Network* [8], C-RAN, est une nouvelle architecture de réseaux visant à améliorer le système présentement utilisé dans les communications mobiles. Nous savons qu'il se dérive des *Radio Access Network* (RAN) traditionnels, qui étaient bâtis avec stations de base (BTS) multiples distribuées géographiquement à travers une région. Cette nouvelle structure vise à rassembler les différents types de traitement de bande de base (baseband 2/3/4G/LTE/5G) dans un même bassin de Station de Base Virtuelle (BS), rendant donc toutes les

actions associées aux protocoles centralisés dans un même centre de données, le « nuage ». L'implémentation d'un tel système sera possible grâce à des concepts comme les « *Network Function Virtualisation* » [9] (NFV) et aux « *Software Defined Networks* » [10] (SDN) qui permettront de virtualiser les actions associées au traitement de protocoles de communication. La réalisation de telles infrastructures aura pour effet de pouvoir réduire les coûts associés à l'implantation de RAN, tels que les frais en capital dits *Capital Expenses* (CAPEX) et les frais d'exploitation dits *Operation Expenses* (OPEX). Ces frais peuvent être diminués grâce au fait que diverses fonctionnalités ne sont plus dédiés mais qu'elles peuvent en fait être partagées dynamiquement.

1.2 Éléments de la problématique

La problématique se compose d'un questionnement à deux étapes. En premier lieu, nous voulons étudier le C-RAN. Nous nous posons donc la question de savoir quelle architecture de plateforme permettrait la mise en place d'un C-RAN pouvant virtualiser le protocole LTE tout en restant flexible et pouvant s'adapter aux nouvelles technologies et services.

Ensuite, nous nous concentrerons sur la gestion des tâches associées à la virtualisation. Les questions de recherche consistent à :

- Déterminer si l'ordonnancement de tâches est faisable sur un simple cœur d'un processeur i7 sous les contraintes de temps imposées par le protocole.
- Identifier quel algorithme d'ordonnancement offre une bonne durée d'exécution et utilisation du processeur.
- Déterminer si un processeur doit être dédié à la gestion de tâches.

1.3 Objectifs de recherche

À travers ce mémoire, deux angles de recherche se distinguent. Le premier a pour but de proposer l'architecture d'une grappe de calcul temps réel pouvant effectuer la virtualisation d'une pile LTE [11]. Cette dernière doit être assez performante pour traiter tous les aspects de la réception et de l'envoi de communications LTE et de versions de protocoles antérieures. De plus, des évaluations

analytiques et expérimentales seront faites sur le système afin d'en avoir une vue déterministe, de s'assurer de sa légitimité et surtout de sa robustesse.

Le deuxième objectif vise à valider des hypothèses en relation avec l'ordonnanceur et l'algorithme d'ordonnancement au sein de notre système d'exécution. Ainsi, nous voulons nous assurer que la gestion des tâches associées au traitement d'une pile LTE est faisable sur un cœur de processeur i7. Puis, nous évaluerons la nécessité de dédier un cœur complet à la gestion des tâches contre le partage de l'ordonnancement sur les unités de calculs disponibles. Ces tests nous permettront aussi de déterminer les algorithmes d'ordonnancement les plus adéquats pour le contexte donné.

1.4 Plan de mémoire

Ce mémoire expose les fruits des efforts fournis tout au long de ce projet de maitrise. Nous commencerons par une mise en contexte avec le chapitre deux. Cette revue de littérature expliquera plus en détail tous les concepts utilisés à travers ce document tout en gardant une vue généraliste. Ensuite, le chapitre trois étudiera les latences et bandes passantes associées à une grappe de calcul temps réel proposé qui permettrait la virtualisation d'une pile LTE. Par la suite, le chapitre 4 présentera les techniques de conception des ordonnanceurs de tâches au sein du système d'exécution StarPU utilisé de manière symbiotique avec la librairie *Data Plane Development Kit* (DPDK). Puis, à travers le chapitre 5, nous verrons les modalités de notre preuve de concept en présentant le plan, la méthode de test et les résultats expérimentaux obtenus. Enfin, nous résumerons les travaux effectués tout au long de cette maitrise et discuterons des résultats obtenus. Plusieurs travaux futurs seront proposés en fin de document.

CHAPITRE 2 REVUE DE LITTÉRATURE

Le corps de ce chapitre servira d'exposition de l'état de l'art sur les sujets traités. Ainsi, cette revue de littérature portera sur trois principaux thèmes : les tâches et algorithmes d'ordonnancement, les microprocesseurs associés aux systèmes d'exécutions (runtime) logiciels, puis les C-RAN et NFV; qui nous permettront de mieux situer le contexte, mais surtout notre contribution, telle que définie dans le chapitre précédent.

2.1 Gestion des tâches associées aux plateformes multicœurs

Dans la section suivante nous définirons ce qu'est l'ordonnancement de tâches, les types d'heuristiques, puis nous finirons par examiner les architectures possibles et la nécessité d'un ordonnanceur matériel.

2.1.1 Ordonnanceur et ordonnancement pour multicœurs

L'ordonnancement est la méthode par laquelle on donne accès aux ressources de la machine aux processus (ensemble de tâches) de calculs d'une application donnée. Le principal but est de pouvoir adéquatement balancer l'utilisation des ressources afin que les processus s'effectuent le plus efficacement que possible.

De cette façon, un ordonnanceur est le mécanisme qui s'occupe de gérer quel processus peut avoir accès aux ressources de la plateforme et quand. Dans notre cas, nous parlerons d'un ordonnanceur de tâches, aussi nommé « task scheduler ».

Dans le contexte LTE un bon ordonnanceur doit tenir compte de contraintes temps réel, car la norme nous impose un temps strict de traitement par usager de 10ms [12] . Ce dernier doit donc prendre en considération :

- Les **échéances** pour assurer que chaque processus respecte la plage de temps qui lui est alloué.
- La **latence**, premièrement le délai d'exécution, c'est-à-dire le temps entre la soumission de la requête et sa complétion. Puis le temps de réponse entre l'envoi de la première requête et la réception de la première réponse (« acknowledge »).
- L'**utilisation** des CPU, donc ne pas surutiliser un processeur alors que d'autres le sont peu.

- Le **débit**, qui correspond au nombre d'exécutions de processus par unité de temps

Afin que l'ordonnanceur puisse gérer les contraintes de temps imposées par le contexte LTE, les tâches seront acheminées avec des descriptifs. Les plus utilisés sont :

- Le **temps de relâchement** (Release time), qui est le temps auquel une tâche est prête à être exécutée.
- Le **délai minimum**, qui est le minimum de temps permis au début de l'exécution d'une tâche et après sa complétion.
- Le **délai maximal**, qui est le maximum de temps permis au début de l'exécution d'une tâche et après sa complétion.
- Le **WCET** (Worst Case Execution Time), qui est le maximum de temps qu'une tâche peut utiliser pour être exécutée.
- Le **Runtime**, qui représente le temps pris pour compléter une tâche sans interruption.
- Le **poids** ou **priorité**, qui est le niveau d'urgence de la tâche.
- La **mémoire consommée**, cela pourrait être la mémoire nécessaire pour exécuter une tâche ou même les espaces requis pour exécuter une fonction reliée à la tâche.

Une fois toutes ces caractéristiques prises en compte, il faut avoir un ordonnancement adéquat, qui nous permettra d'atteindre la qualité de service voulue.

2.1.2 Algorithmes d'ordonnancements

Dans les domaines du « *Cloud Computing*» et du « *High Performance Computing* (*HPC)* », un ordonnancement efficace des tâches d'un processus est primordial afin d'avoir la plus grande performance possible». Il existe une abondante littérature sur le sujet de l'ordonnancement de tâches, ce que nous pouvons en retirer est qu'il n'existe pas une solution parfaite qui a la capacité de résoudre tous les problèmes. Ainsi, au lieu de s'efforcer à trouver l'ultime solution d'ordonnancement, les concepteurs se fient à des algorithmes ou stratégies, qu'ils choisissent en fonction des caractéristiques de l'application. Il existe de nombreux algorithmes d'ordonnancement, la figure 2-1 montre la classification des ordonnanceurs de tâches [7].

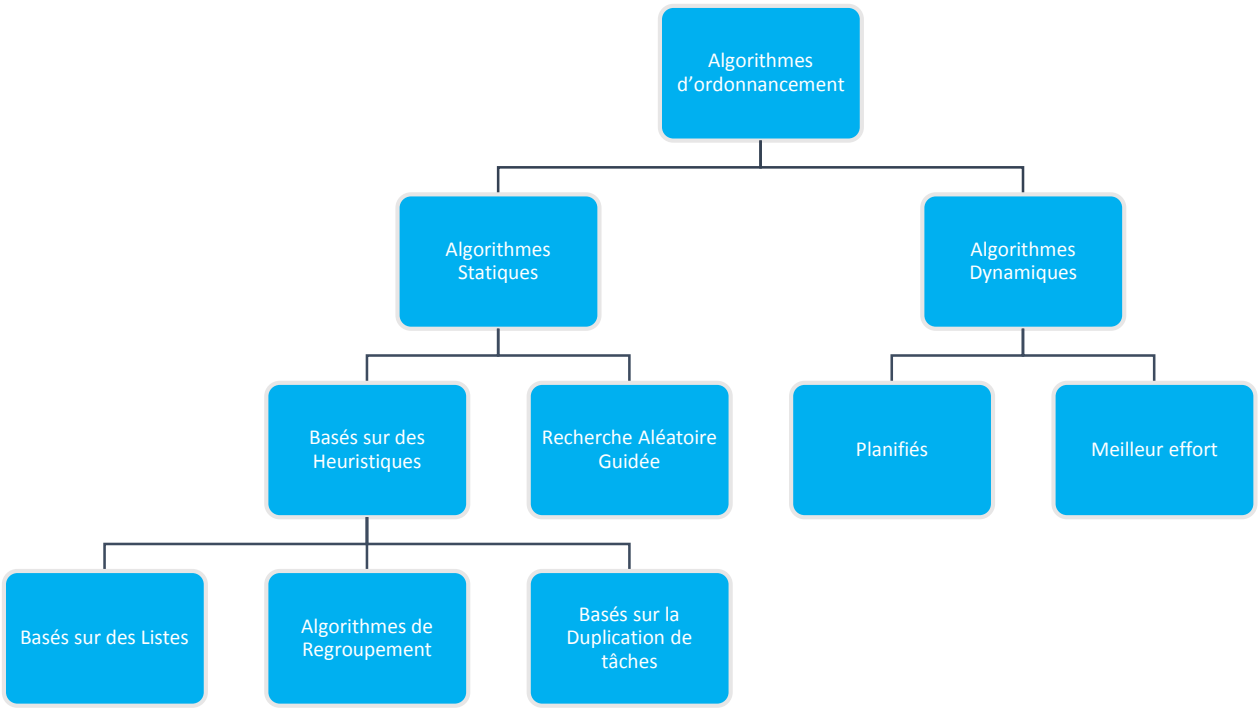

Figure 2-1 Classification des algorithmes d'ordonnancement

2.1.2.1 Algorithmes statiques

Les algorithmes statiques, aussi surnommés déterministes sont sans doute les algorithmes les plus utilisés et étudiés. Bien que relativement simples à implémenter ils manquent souvent de flexibilité. Ils sont reconnus pour mieux gérer les tâches périodiques que les apériodiques, cependant il existe de nombreuses méthodes qui permettent de mieux adapter ces algorithmes pour le traitement de ces types de tâches. Par contre, les algorithmes statiques ont deux principaux désavantages, une faible utilisation des processeurs et la mauvaise gestion des tâches apériodiques, mais ils ont extensivement été étudiés à travers la littérature.

Le point commun des algorithmes statiques est qu'ils peuvent avoir une ambivalence à la préemption. Une caractéristique partagée avec de nombreux systèmes temps réels est que l'ordre d'exécution des tâches est connu a priori et comporte de nombreuses dépendances de tâches. Ces genres de systèmes peuvent être ordonnancés de manière non préemptive, ce dernier permet

d'éviter les coûts ajoutés dus à des changements de contexte. Les types d'ordonnanceurs statiques les plus utilisées dans le contexte des systèmes multiprocesseurs sont :

Le **Eager** [13], communément appelé glouton, est constitué d'une liste de tâches prêtes à être exécutés en ordre de priorités ou de dépendances résolues pour laquelle tous les processeurs se disputent l'accès aux tâches. Ainsi, dès qu'un processeur est libre, il essaie de se procurer une tâche comme le montre la figure 2-2.

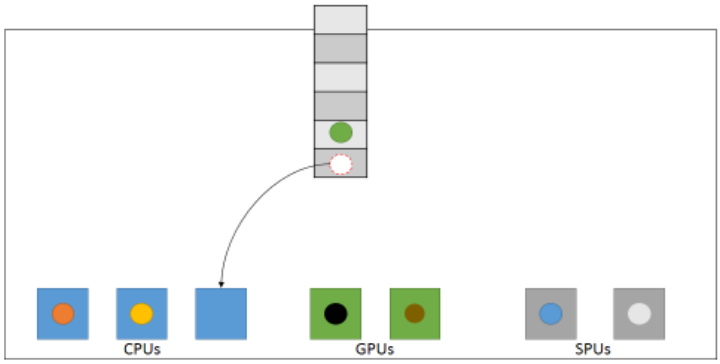

Figure 2-2 Algorithme Glouton

Le **Work Stealing** [14], où chaque processeur possède sa liste de tâches à exécuter, mais dans l'éventualité où un processeur finit toute sa liste de tâches, il ira voler une tâche à exécuter de priorité moindre dans la liste d'un autre processeur, illustré dans la figure 2-3.

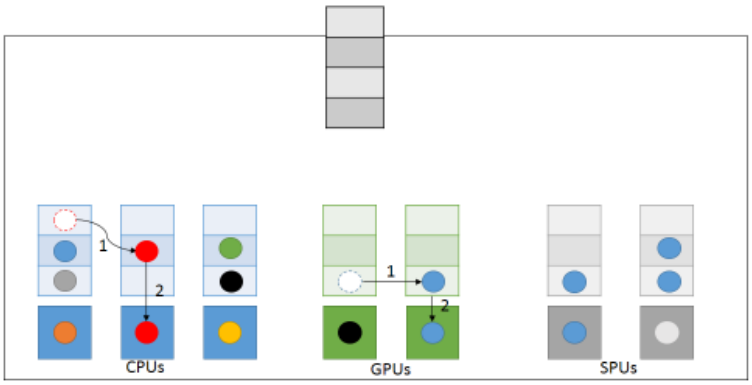

Figure 2-3 Algorithme de Work Stealing

Le **Priority Scheduler** [15], où il existe plusieurs listes de tâches prêtes à être exécutées, organisées par leurs niveaux de priorité. Ainsi les listes à priorités plus élevées doivent être consommées en premier. Cette méthode d'allocation est illustrée à la figure 2-4 :

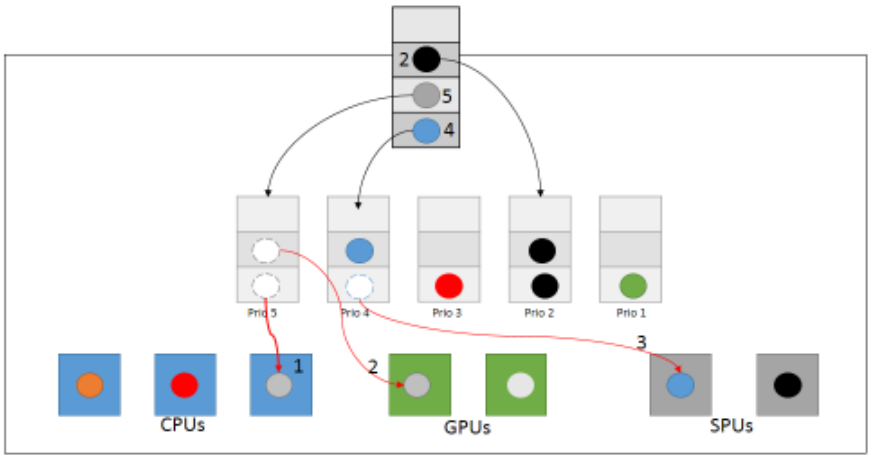

Figure 2-4 Ordonnanceur par priorités

Bien que couramment utilisés et peu complexes, ces algorithmes sont considérés comme rigides et résultent souvent en des performances médiocres pour de nombreuses applications avec des contraintes de temps plus rigides. Il faut donc se tourner vers d'autres types d'algorithmes d'ordonnancement. Dans la littérature on retrouve deux grandes familles d'algorithmes statiques.

Premièrement, nous avons les algorithmes d'ordonnancements basés sur une **recherche aléatoire guidée**. Ils utilisent des techniques de recherche aléatoire afin de naviguer à travers un problème, cependant il ne faut pas confondre avec des choix de manière aléatoire à travers un problème donné. Ces techniques se basent sur l'information obtenue à travers les résultats recherchés précédemment combinés avec des composantes de hasard afin de générer de nouveaux résultats possiblement plus adéquats. Parmi les algorithmes basés sur la recherche aléatoire guidée, les **algorithmes génétiques** (AGs) [16] sont les plus populaires et les plus utilisés dans des domaines variés. Ils permettent de trouver une solution adéquate dans un large espace de solution en un temps polynomial en se basant sur des techniques évolutives découlant de la nature. L'AG commence avec une population initiale d'individus, générés aléatoirement ou grâce à un algorithme connexe. Chaque individu est constitué d'un ensemble de paramètres (bagage génétique) permettant d'identifier de manière similaire une solution potentielle à un problème. Ainsi, dans chaque génération la population passe à travers des mutations, des processus de croisement et de sélection. À travers ces croisements et mutations, des parties d'individus de la population sont échangées et créent de nouveaux individus qui remplacent ceux à partir desquelles ils ont évolué. Chaque individu est sélectionné pour un croisement grâce à une probabilité de taux de recouvrement. La mutation modifie un ou plusieurs gènes (les paramètres) dans un chromosome

avec une probabilité de taux de mutation. Ce phénomène d'évolution représente les algorithmes de types génétiques. Il faut noter que pour avoir une solution optimale pour un contexte donné, il faut définir les paramètres de contrôle des AG précisément, mais ces derniers ne seront pas forcement compatible avec tous les types de contextes. Les AG permettent d'avoir un bon ordonnancement de tâches, mais le temps d'ordonnancement est souvent plus élevé que ceux basés sur des heuristiques. En addition aux AG, le recuit simulé (basé sur le processus physique de recuit, qui est le processus thermique qui permet d'obtenir les états de faible énergie cristalline d'un solide), la méthode du A* (qui vise le meilleur au premier, il est originalement du domaine de l'intelligence artificielle), la recherche tabu (qui tente d'évité les minimaux locaux et guide la recherche vers un minimum global) et d'autres méthodes se retrouvent aussi dans ce groupe de méthodes basées sur une recherche aléatoire guidée.

D'autre part, dans le groupe des algorithmes statiques [17] , nous retrouvons le groupe d'algorithmes les plus utilisés, ceux basés sur des heuristiques. Ce dernier peut être classifié en trois différents sous-groupes d'heuristiques basées sur : des listes, des regroupements et la duplication de tâches.

Une heuristique **basée sur des listes** [18] comme son nom l'indique, maintient une liste de toutes les tâches d'un contexte ordonné par rapport à leurs priorités. Elle fonctionne en deux phases : la *sélection de tâches*, où elle remplit une liste avec les tâches à priorités les plus élevées prêtes à être exécutés et la phase de *sélection de processeur*s, qui choisit le processeur le plus adéquat pour exécuter ces tâches, dans un but de minimiser un coût prédéfini. L'ordonnanceur choisit donc l'ordre le plus adéquat de tâches grâce à une heuristique. Deux heuristiques communément utilisées dans le domaine des multiprocesseurs hétérogènes sont :

Le **Critical Path to Processor** (CPOP) [6], qui comporte un peu comme dans le cas du work stealing, une liste associée à chaque unité de traitement. Lors de l'éventualité de plusieurs processeurs libres, l'heuristique choisira où acheminer une tâche en fonction du temps de communication avec un des processeurs. Ainsi, celui avec le temps le plus court remporte l'exécution de la tâche, comme le montre la figure 2-5. Cette méthode est communément utilisée, mais elle est sujet à la plateforme sur laquelle sont exécutées les tâches. Ainsi, plus la plateforme est performante et que les temps de communications sont faible, moins l'heuristique est efficace.

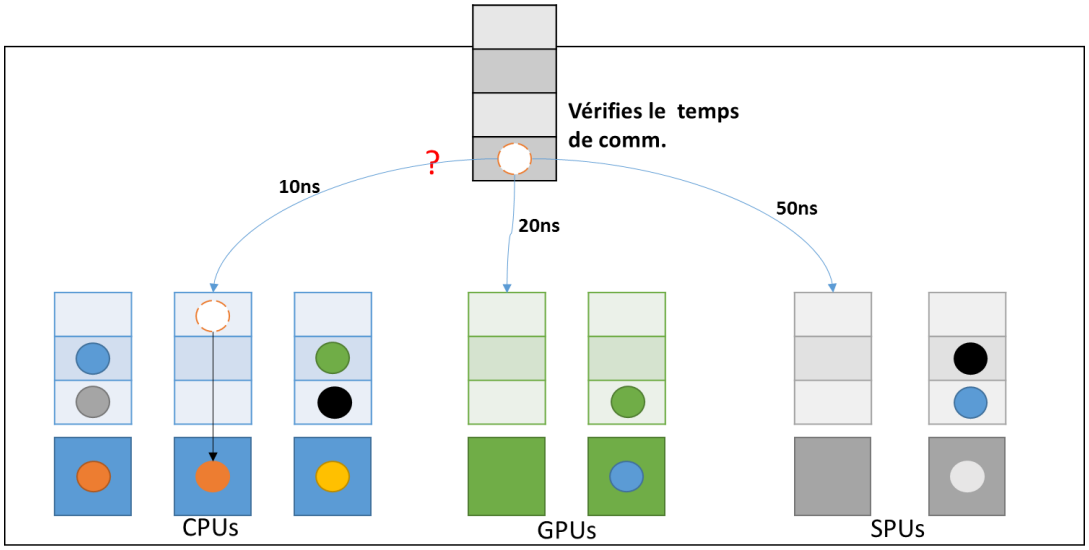

Figure 2-5 Algorithme CPOP

Le **Heterogeneous Earliest Finish Time (**HEFT**)** [6], [19], [20] utilise aussi une queue de tâches associée à chaque processeur, cependant lors de l'assignation de ces dernières, l'heuristique calcule et compare quand la tâche finira plus tôt sur chacune des unités de calcul. L'algorithme assigne la tâche en fonction de cette prévision, comme expliqué sur la figure 2-6. Cette heuristique a l'avantage de donner un temps d'exécution de toutes les tâches très court, mais elle peut résulter en une utilisation de processeurs non optimale.

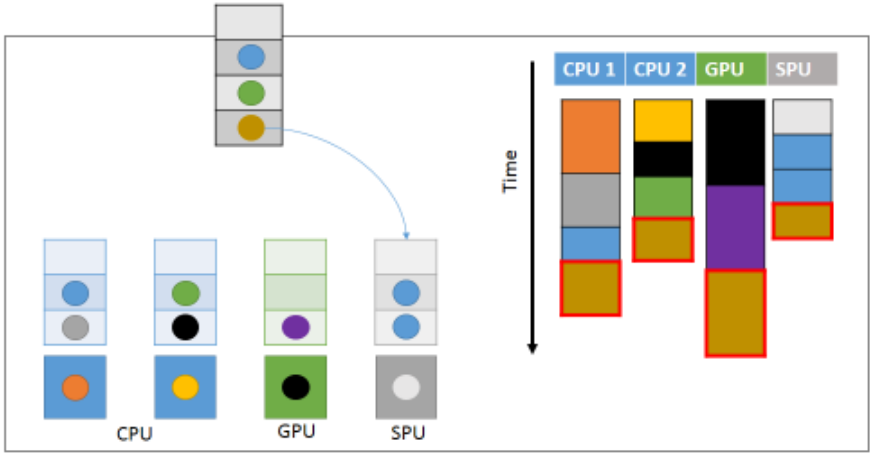

Figure 2-6 Algorithme HEFT

Il existe de nombreuses autres heuristiques basées sur des listes tel que, le **Modified Critical Path** (MCP) [21] qui donne la plus haute priorité à une tâche qui possède le temps de début (Start

time) le plus court. Le principal inconvénient de cette méthode est qu'elle n'inclut pas les temps de communication lors du traitement des priorités des tâches. Pour finir, les heuristiques basées sur des listes, **l'Insertion Scheduling Heuristic** (ISH) [22] qui se servent des temps de repos créés par des utilisations partielles des processeurs pour y insérer des tâches, sont considérés comme des algorithmes efficaces. Les heuristiques basées sur des listes sont généralement plus pratiques et démontrent plus de performance au niveau du temps d'ordonnancement par rapport aux autres groupes.

Les heuristiques **basées sur les regroupements** de tâches ont pour principale fonctionnalité de réunir les tâches en groupes. Une fois tous les groupes formés, ils peuvent être de différentes tailles et assignés à différents processeurs. L'algorithme raffinera l'ordonnancement au fur et à mesure en fusionnant les différents groupes de tâches. Si des tâches font partie d'un même groupe, elles devront être exécutées sur un même processeur. Afin qu'un algorithme d'ordonnancement de type regroupement soit final, il devra passer par plusieurs étapes supplémentaires par rapport aux autres types d'heuristiques. Tout d'abord, il y a l'étape du fusionnement des groupes, une étape qui devra se répéter tant et aussi longtemps que le nombre de groupes n'est pas équivalent au nombre de processeurs. Puis, une fois les groupes assignés à des processeurs, l'algorithme devra ordonnancer les tâches dans chaque groupe pour être exécutés sur le processeur en question. Les algorithmes basés sur le regroupement les plus communs sont :

Le **Triplet** [20], [23], qui analyse les tâches par groupe de trois. Cet algorithme possède par ce fait trois phases. La première, Task Clustering Phase, qui groupe les tâches en triplets en fonction de temps de communication vers un processeur. Ensuite, l'algorithme regroupe les processeurs ayant des caractéristiques similaires pendant la Processor Clustering phase, rendant ainsi les groupes plus homogènes. Puis, la dernière phase consiste à assigner ces groupes de tâches à des groupes de processeurs. L'avantage de cette heuristique est qu'elle capitalise sur la réduction des temps de communications entre tâches. Cependant cet algorithme peut être complexe à implémenter et il possède un temps d'exécution élevé sans garantie d'améliorer le temps de traitement des tâches.

Le **Dominant Sequence Clustering** (DSC) [24] analyse un contexte de tâches et détermine quelles tâches sont considérées comme dominantes (DS). Une DS est une séquence qui comprend la tâche maitresse, ou un groupement de tâches dépend d'une tâche. Ainsi l'algorithme mettra ces

séquences dominantes dans de mêmes groupes, puis les mappera vers un processeur, respectivement.

Il existe de nombreuses autres heuristiques basées sur des listes tel que la **Linear Clustering Method** [7], la **Mobility Directed Method** [7] et l'heuristique au cœur du **Clustering and Scheduling System**(CASS) [7]. Le grand avantage des heuristiques basées sur le regroupement est qu'elles nous permettent de réduire le temps de communication entre tâches, car les tâches similaires et dépendantes les unes des autres sont souvent regroupées et exécutées sur un même processeur. Cependant, cet avantage vient avec un compromis assez important; l'activité de regroupement peut induire un ordonnancement non optimal lorsque le temps d'exécution est une contrainte solide. De plus, ce type d'algorithme est souvent peu efficace au niveau de l'utilisation maximale des processeurs.

Pour finir avec les algorithmes statiques, il y a les heuristiques **basées sur la Duplication de tâches**. La stratégie est d'ajouter de la redondance de tâches afin de réduire les communications interprocessus, due à des tâches dépendantes, mais exécutées sur des processeurs différents. Il existe de nombreux algorithmes de ce type, tels que le **Critical Path Fast Duplication** [7], **Duplication Scheduling Heuristic**, **Bottom-Up Top-Down Duplication Heuristic** [22] et le **Duplication First and Reduction Next**. Ainsi ces types d'heuristiques sont avantageuses dans le contexte d'infrastructures avec de nombreux processeurs et des jeux de tâches ou beaucoup de sous-tâches sont dépendantes d'une tâche « maitresse ». Le principal inconvénient est que ces algorithmes possèdent des complexités élevées par rapport aux autres types d'heuristiques statiques.

2.1.2.2 Algorithmes dynamiques

Les algorithmes dynamiques, aussi surnommés non déterministes sont sans doute les algorithmes qui demandent le plus de ressources. Cela les rend souvent complexes à implémenter, mais leur permet d'avoir une très grande flexibilité. Ils sont reconnus pour mieux gérer les tâches apériodiques et sporadiques, tout en maintenant un bon rendement avec des tâches périodiques. Il existe de nombreuses méthodes permettant de réduire les ressources nécessaires lors de l'ordonnancement, mais cela ne fait que rajouter à la complexité des algorithmes en question. Les algorithmes dynamiques se divisent en deux sous-groupes, ceux basés sur la planification et ceux basés sur le meilleur effort.

Les algorithmes basés sur la **Planification**, déterminent lors de la réception d'une tâche, si cette dernière peut être traitée dans les contraintes de temps établies. Dans le cas où elle entre dans les contraintes mises en place, la tâche sera exécutée par un processeur, sinon elle sera rejetée. Si la tâche rencontre son échéance sans interférer avec la complétion d'une autre tâche, elle sera tout de même acceptée dans le plan d'ordonnancement. Ne pas rencontrer une de ces contraintes préétablies, résultera en le rejet de la tâche. Les algorithmes dynamiques basés sur la planification sont généralement très flexibles, surtout lorsqu'il en tient au traitement de tâches à caractère apériodique et ce tout en garantissant la complétion de tâches avec des contraintes de temps serrées. L'avantage de ces algorithmes est qu'ils offrent un pourcentage d'utilisation des processeurs plus élevé que les algorithmes statiques tout en garantissant la complétion des tâches passant les critères de l'algorithme. Les plus utilisés sont le **Dynamic Priority Exchange**, le **Dynamic Sporadic** et l'**Improved Priority Exchange**, tous basés sur l'**Earliest Deadline First** [25] .

Dans le cas des algorithmes basés sur le **Meilleur Effort**, il n'y a pas de test d'acceptation des tâches. Toutes les tâches sont prises pour être ordonnancées et le meilleur effort est fait pour essayer de rencontrer les contraintes de temps. En général, si le système s'aperçoit qu'il sera incapable de rencontrer les contraintes imposées sur une tâche, il ne fera que la rejeter. Ainsi il n'y a aucune garantie que les tâches seront exécutées à temps. L'avantage de ces types d'algorithmes est qu'ils ont une complexité moins élevée que ceux basés sur la planification et ils possèdent quand même un meilleur pourcentage d'utilisation des processeurs que les algorithmes statiques. Donc dans le contexte des algorithmes dynamiques, ceux basés sur le meilleur effort offrent une alternative compatible avec plus de cas et une complexité moindre que les algorithmes basés sur la planification. Les deux algorithmes basés sur le meilleur effort les plus utilisés sont le **Dependant Activity Scheduling Algorithm** [19], et le **Lockes Best Effort Scheduling Algorithm** [19]**.**

2.1.2.3 Ordonnanceur matériel

Comme énoncé dans les parties précédentes, il existe de nombreux algorithmes d'ordonnancement efficaces. Une fois la décision faite sur le type d'ordonnancement voulu, il faut décider comment implémenter ce dernier. Afin d'évaluer la meilleure solution, nous devons situer le problème.

À travers la dernière décennie, les processeurs ont considérablement augmenté la cadence d'horloge. Une fois avoir atteint la cadence la plus élevée possible compte tenu des contraintes de densité de puissance, il a fallu remédier à la quête de performances accrues par un autre moyen.

Ceci a motivé l'émergence des multi/many cores. Cependant, l'ajout de plus de processeurs et de cœurs fait croître aussi la consommation en puissance. Une solution possible apparaît avec les plateformes hybrides, où des blocs matériels dédiés accélèrent et augmentent la puissance de calcul des systèmes multicoeurs avec une consommation de puissance modérée.

Ainsi l'implémentation matérielle d'un ordonnanceur peut permettre de nous affranchir de l'aspect séquentiel des processeurs. Il est donc possible que l'implémentation d'un même algorithme soit accéléré par une implémentation en matériel de par le fait qu'elle peut prendre avantage d'une plateforme parallèle. De plus, le fait d'avoir un accélérateur matériel permet de sauver du temps d'utilisation du CPU [26], nous permettant d'avoir une meilleure utilisation des ressources. Enfin, un dernier avantage associé à ce type d'ordonnanceur est qu'il nous permet d'optimiser le temps d'exécution des applications. Il faut savoir que lorsqu'une application, en C par exemple, est lancée sur une machine, la tâche à accomplir doit non seulement effectuer les calculs nécessaires au niveau de l'application, mais une fois prête à être exécutée, l'OS doit passer par d'autres applications pour allouer les espaces mémoire nécessaires, passer par des pilotes et d'autres couches logicielles afin de pouvoir être exécutée. Un accélérateur matériel nous permet de court-circuiter toutes ces étapes et d'attribuer directement la tâche aux processeurs en utilisant les accès matériels de la machine. Notons que cela est dépendant de l'architecture du système.

La plupart des ordonnanceurs matériels sont basés sur des algorithmes statiques utilisant des listes, car cela nous permet d'avoir une architecture simplifiée. Pour représenter ces listes, les ordonnanceurs utilisent fréquemment des registres à décalage (SR), des mémoires souvent implémentée dans des modules mémoire appelés 'Block Ram' (BRAM) et de la logique combinatoire. L'efficacité d'un ordonnanceur matériel est donc dépendante de l'algorithme implémenté et des instances utilisés dans l'architecture de ce dernier. Un exemple simple d'implémentation serait l'utilisation d'une mémoire externe versus une chaine de registre. Bien sûr, une mémoire externe permet de stocker plus de données qu'un mémoire interne dans un ASIC ou un FPGA, mais elle induit une latence significative par rapport aux chaines de registres intégrée dans une puce.

Le principal désavantage d'un ordonnanceur matériel [27] découle de la complexité d'introduire ces blocs matériels dans un système temps réel. Il faut gérer les latences afin de rester dans les contraintes de temps. Il faut aussi gérer les accès au CPU par la plateforme matérielle et

cela induit un important travail d'intégration et de synchronisation. De plus, l'implémentation de certains algorithmes sur une plateforme matérielle peut s'avérer beaucoup plus complexe que sur CPU, où nous avons accès à de nombreuses bibliothèques fournissant des blocs de base utiles.

2.2 Multiprocesseurs et systèmes d'exécution logiciels

La loi de Moore [28] découle d'une observation à l'effet que le nombre de transistors dans un circuit intégré double tous les 2 ans. Suite à cette observation, l'industrie écrivant sa propre prophétie, nous avons obtenu les principes de « More Moore »; exposant la miniaturisation des fonctions numériques; et même « More than Moore » [28] ; exposant la diversification fonctionnelle des systèmes numériques.

Cependant, nous observons une stagnation de la fréquence d'opération des processeurs en raison des contraintes de dissipation de puissance et du problème de courants de fuites, qui a donc poussé les fournisseurs de matériel à augmenter le parallélisme dans leurs processeurs afin d'améliorer les performances des applications scientifiques et d'ingénierie hautement parallèles.

Cela conduit à une ère de l'informatique hétérogène où accélérateurs parallèles sont jumelés avec des processeurs compatibles au x86.

Comprendre les performances de ces systèmes informatiques hétérogènes est devenu très important, car ils ont commencé à apparaître dans les plateformes de calcul scientifique et de l'ingénierie d'extrême échelle.

Les microprocesseurs d'aujourd'hui ont des sous-systèmes de mémoire complexes avec plusieurs niveaux d'antémémoire. L'utilisation efficace de cette hiérarchie de mémoire est essentielle pour obtenir des performances optimales, en particulier sur les processeurs multicoeurs.

2.2.1 Les microprocesseurs

Le microprocesseur est à la base un circuit intégré qui a pour but d'exécuter des instructions et de traiter des données. De nos jours, il constitue le centre de l'ordinateur, il sert de plateforme de calcul programmable polyvalente. En général, un processeur reçoit des données numériques en entrée, les traite selon des instructions stockées dans sa mémoire, et fournit des résultats en sortie. Un processeur diffère d'un autre par rapport aux caractéristiques suivantes :

- Le **nombre de bits** qu'il peut traiter à la fois. Un processeur d'ordinateur gère 32 ou 64 bits si nous prenons le cas des processeurs d'Intel, mais il en existe des 4-8 et même 16 bits.
- Le **jeu d'instruction**, tel qu'ARM, PowerPC et x86 dans le cas d'Intel.
- La cadence d'horloge, elle sert à réguler le rythme de travail du processeur, ainsi une cadence plus élevée signifie une exécution d'instructions plus rapide.
- Mais surtout son architecture, qui est la spécifie l'organisation d'un processeur.

Comme exposé plus haut, les processeurs suivent la loi de Moore, cependant les fréquences d'opération stagnent. L'industrie s'est donc tournée vers la parallélisation, c'est-à-dire intégrer sur une même puce plusieurs cœurs de processeurs, d'où la venue des processeurs multicoeurs.

Ainsi un processeur multicœurs contient de multiples unités de calculs proches les unes des autres, et elles peuvent communiquer entre elles à haute vitesse par le biais d'un bus.

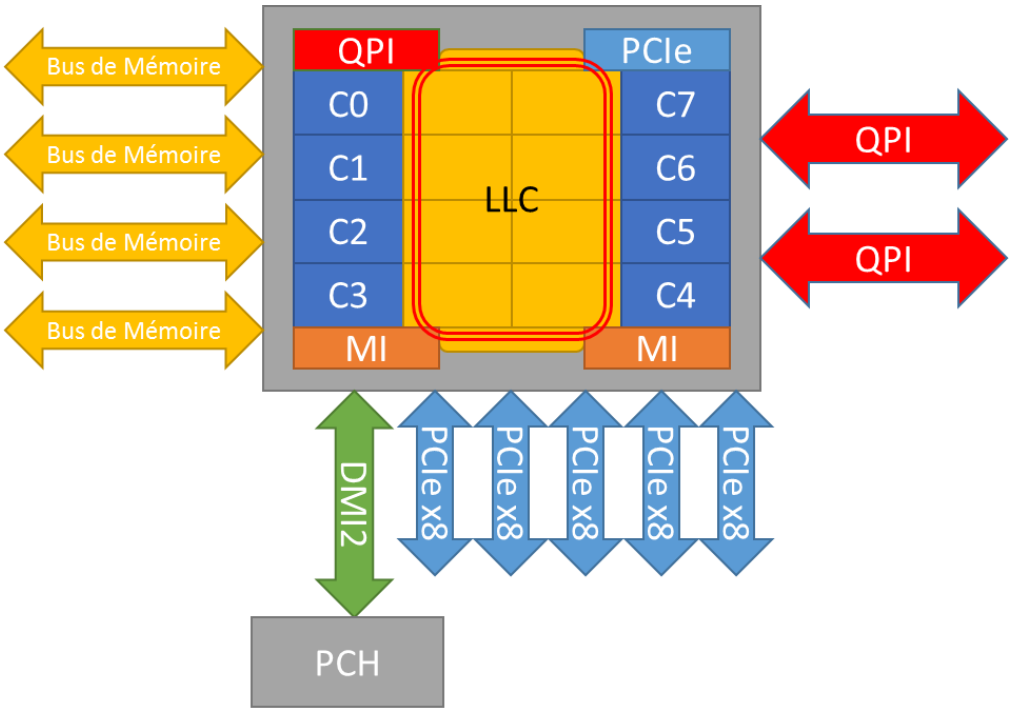

Figure 2-7 Architecture d'un processeur multicoeurs

Ces derniers ont donc pour avantage de pouvoir exécuter plusieurs tâches en même temps, mais surtout de le faire avec moins d'énergie qu'une architecture qui aurait été obtenu par un

prolongement des technique en vigeur il y a quelques années. Auparavant, pour avoir une unité de calcul plus performante, il fallait augmenter sa cadence d'horloge, de par ce fait, un processeur consomme plus de courant, ce qui causerait des problèmes de surchauffe. Alors un multicoeur permet plus de performance avec plusieurs cœurs cadencés a une fréquence plus basse, offrant donc plus de flexibilité et des économies d'énergie.

Replaçons le sujet dans le contexte de ce mémoire : le calcul de haute performance. Toutes les grappes de calcul emploient des processeurs multicoeurs. Elles sont basées sur deux principaux types d'architectures, celles basées sur un faible nombre de cœurs et celles basées sur un nombre élevé de cœurs.

2.2.1.1 Les MICs

Les « Many Integreted Cores » [3], aussi appelés « Many Cores » contiennent un nombre élevé de cœurs. Un MIC typique contient 64 ou même 100 cœurs cadencés à une fréquence modérée aux environs de 1.2 GHz.

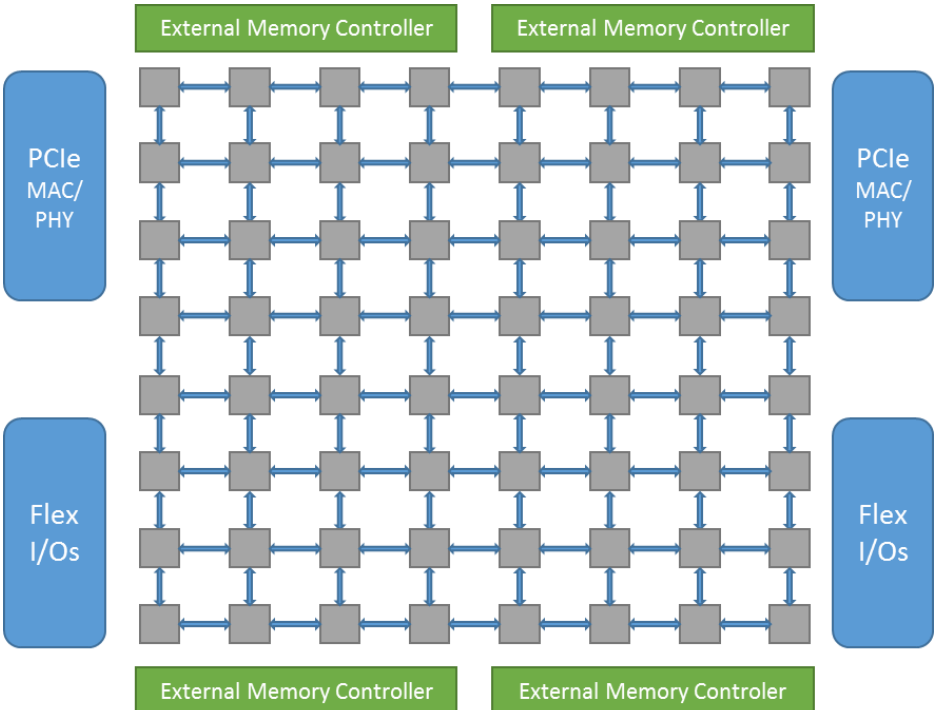

Figure 2-8 Architecture d'un MIC 64 cœurs

Si nous regardons les performances brutes, c'est-à-dire en instructions exécutées par unité de temps, plus de cœurs moins puissants sont plus efficaces que moins de cœurs exploités à cadence plus élevée. L'architecture MIC tend donc à prévaloir lors de calculs hautement parallèles tel que

rencontrés dans une opération de type « scatter / gather » qui offre des capacités de streaming nécessitant une haute bande passante au niveau de la mémoire, afin de pouvoir gérer une grande quantité d'information rapidement.

Comme énoncé au-dessus l'architecture sur laquelle se base le processeur est importante, car elle permet d'avoir des fonctionnalités spécifiques pouvant être bénéfiques pour le domaine du « *High Performance Computing* » (HPC). Prenons pour exemple le Xeon Phi 5110P de Intel [13]. Il possède 60 processeurs fonctionnant à une cadence d'opération de 1.05 GHz tous connectés les uns aux autres via un bus de communication « Full Duplex ». Le Phi étant conçu par Intel, il se base sur une architecture x86 avec un jeu d'instruction de 64 bits au niveau de l'adressage, mais il contient surtout des vecteurs d'instructions et des registres de 512 bits SIMD de large. Cela signifie qu'un noyau avec une antémémoire L1 de 32 Ko pour les instructions et les données L1 et une antémémoire L2 de 512 KB unifiée par cœur, peut exécuter deux instructions par cycles d'horloges.

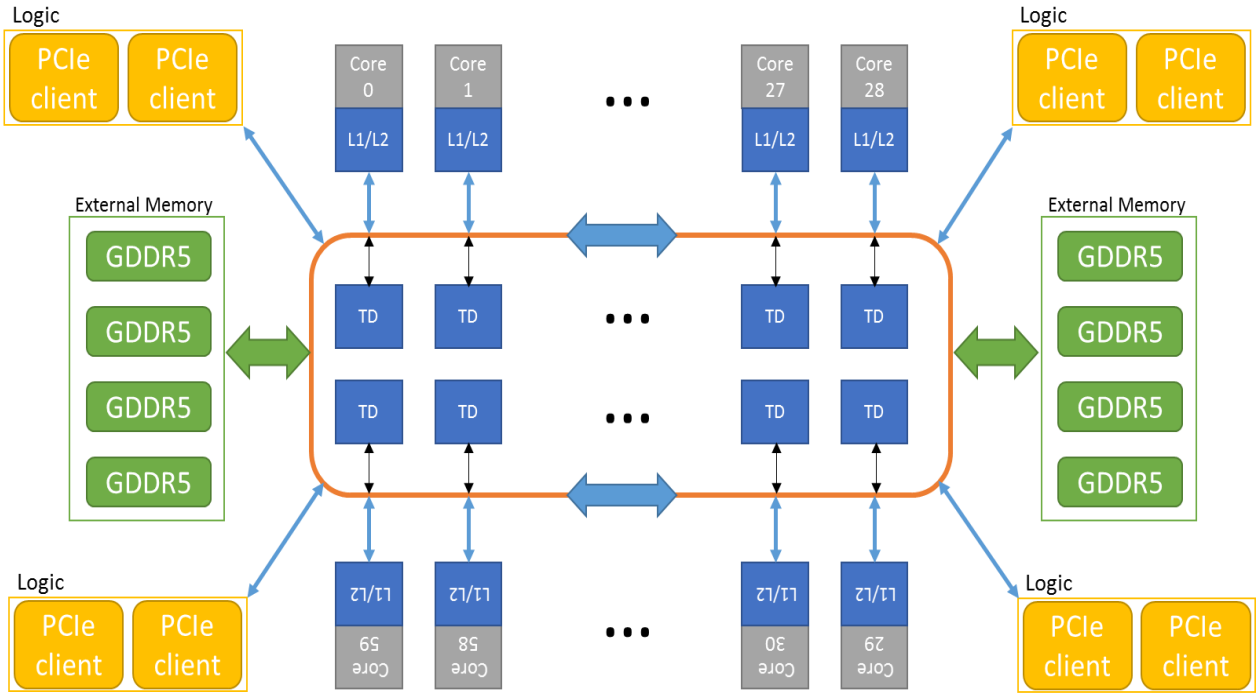

Figure 2-9 Architecture d'un Xeon Phi

Afin de garder une cohérence minimale entre les blocs d'antémémoires associés à chaque cœur, le Phi possède des « Tag Directory (TD) » distribués uniformément sur le bus. Notons que chaque noyau comprend 4 fils matériels (*threads*) pour un total de 240, le tout est jumelé à de la mémoire

externe de type GDDR5 (Graphics Double Data Rate) qui font de ce coprocesseur une plateforme dédiée à la parallélisation.

Nous observerons dans la partie suivante, deux microprocesseurs typiques des HPCs basées sur un faible nombre de cœurs. Plus loin dans le chapitre, nous présenterons une comparaison des spécificités techniques de chacun des types de processeurs.

2.2.1.2 Les multiprocesseurs

Les multiprocesseurs sont similaires aux MIC, mais sont constitués de moins de cœurs exploités à cadences plus élevées. Leur architecture est similaire à celle décrite par la figure 2-7. Ils comportent en général entre 8 et 12 noyaux cadencés entre 3 et 4 GHz sur une même puce, avec deux niveaux d'antémémoires dédiés à chaque cœur appelés, L1 et L2, et un niveau partagé entre tous les coeurs, L3. Le partage d'antémémoire permet de garder une cohérence plus élevée qu'avec des MIC, rendant l'utilisation de multicoeurs plus adéquate pour des applications contenant beaucoup de dépendances de tâches ou de données. La présence de moins de noyaux permet à chacun d'entre eux d'être plus puissants tout en restant sous un seuil de consommation d'énergie nominal. Nous comparons donc les spécifications techniques de deux leaders dans le domaine des multiprocesseurs dans l'industrie, le Xeon E5 architecture Sandy Bridge [29] et le POWER8 de IBM [30] dans le tableau suivant.

De par le tableau 2-1, nous comprenons que les multiprocesseurs visent des charges de travail devant être exécutées sur des serveurs. Particulièrement, pour des applications « mono filaires » tout en étant soucieux de la consommation d'énergie. Un bon exemple d'application mono filaire, serait l'encodage de fichiers audio, tel que LAME, qui tout au long du traitement d'un fichier, n'utilise qu'un fil.

Sur papier, l'IBM POWER8 semble éclipser la compétition [31] . Il est 25% plus rapide par coup d'horloge que le Xeon E5, il possède plus de trois fois d'antémémoire sur puce et 25 % plus d'accès à de la mémoire externe. Ajoutons que le POWER8 peut ajouter un niveau d'antémémoire L4 de 128 MB hors puce. Sans parler de l'efficacité, avec ses 12 cœurs à 4.15 GHz supportant 8 fils chacun, il peut exécuter 96 fils par cycle d'horloge, contre juste 16 à 2.60 GHz à la base mais avec une capacité d'avoir un Turbo momentané pour le Xeon E5. Pour ajouter insulte à l'injure, le processeur d'IBM comprend un bus de communication similaire au QPI d'Intel, le CAPI (« Coherent Accelerator Processor Interface ») qui contrairement au précédent n'est pas propriétaire,

donc totalement ouvert. Le principal inconvénient du POWER8 est donc sa consommation de puissance, car autant de cœurs cadencés à une fréquence si élevée consomment environ 250 W de puissance contre 115W pour le Xeon E5. La décision de quel processeur utiliser dans un serveur ciblant le HPC revient à l'utilisateur et est dépendant de l'application traitée.

Tableau 2-1 Comparaison des processeurs Xeon E5/ Phi et du IBM Power8

	Intel Xeon E5-2670	**IBM Power8**	**Intel Xeon Phi**
Processeur			
Architecture	Sandy Bridge	Power Architecture	Many Integrated Core
Nombre de cœurs/ processeurs	8	12	60
Fréquence de base (GHz)	2.60	4.15	1.05
Fréquence Turbo (GHz)	3.20	N/A	N/A
Calculs virgule flottante/ horloge	8	N/A	16
Performances/ cœur (Gflop/s)	20.8	N/A	16.8
Performances totales (Gflop/s)	166.4	N/A	1008
Largeur de vecteur SIMD	256	256	512
Nombre de fil/cœurs	2	8	4
Multithreading (on/off)	On ou Off	On ou Off	Toujours On
Type de Multithreading	HyperThread	Oui	Fil matériel
Contrôleur E/S	Sur Puce	Sur Puce	N/A
Mémoire Cache			
Taille de L1 / cœur	32 KB (I) + 32KB (D)	32 KB (I) + 64KB (D)	32 KB (I) + 32KB (D)
Taille de L2 / cœur	256 KB	512 KB	512 KB
Réseau de L2	Anneau Bidirectionnel	N/A	Anneau Bidirectionnel
Taille de L3	20 MB (Partagée)	96 MB (Partagée)	N/A
Réseau de L3	Anneau Bidirectionnel	N/A	N/A
Nœud			
Nombre de processeurs par nœuds	2	2	2
Bande passante du QPI	8.0 GT/s	N/A	N/A
Nombre de QPI	2	N/A	N/A
Bande passante du CAPI	N/A	230 GB/s	N/A
Nombre de CAPI	N/A	N/A	N/A
Type de Mémoire	4 x DDR3-1600MHz	32 x DDR3	8 x GDDR5-3400MHz
Mémoire/ nœuds (GB)	32	32	16-8/ Carte Phi
Socket-Socket interconnecté	2 x QPI, 8 GT/s	PCIe 3.0	N/A

Tableau 2-2 Comparaison des processeurs Xeon E5/ Phi et du IBM Power8 (suite)

Host-Phi Interconnecté	**PCIe**	**PCIe**	**PCIe**
PCI Express	40 PCIe 3.0	11 PCIe 3.0	16 PCIe 2.0
Vitesse du PCIe (GT/s)	8 (GT/s)	8 (GT/s)	5 (GT/s)

Une fois le processeur choisi, il faut avoir un moyen de pouvoir les utiliser à notre guise et ce avec le plus de contrôle. C'est ici qu'arrivent les systèmes d'exécution logiciel communément appelés '*runtime systems*' ou '*runtime environments*'. La section suivante fera une revue des systèmes d'exécution les plus communément utilisés dans le domaine.

2.2.2 Les systèmes d'exécution

Pour soutirer le meilleur rendement possible des processeurs multifilaires (*multithreaded*), les systèmes d'exécution seuls ne font pas l'affaire. Ils doivent être couplés avec un type de programmation qui profite de l'architecture du matériel. Il s'agit d'une programmation basée sur les tâches, le format de tâches et d'ordonnanceur de tâches sera étudié dans la section 2.2 (Tâches et Ordonnanceurs).

Dans l'optique où la performance d'un système est primordiale, penser en terme de fil (*thread*) est inefficace [5]. Il est préférable d'écrire un code sous la forme de tâches logiques pour 5 principales raisons.

Premièrement, ce paradigme est préférable lors du jumelage du parallélisme avec les ressources disponibles. Par exemple, dans un cas idéal où un fil logique correspond à un fil matériel, un est associé à l'autre et l'application roule sans embuche. Par contre, s'il y a plus de fil logique (virtuels) que de fils matériels disponible, l'application peut tomber dans des cas de sous et sur inscription, résultant à des pertes de performances. Alors qu'avec l'utilisation de tâches, l'ordonnanceur s'occupe de les associer avec les fils matériels disponibles.

Deuxièmement, les tâches étant plus légères qu'un fil logique, elles passent moins de paramètres et sont en général non préemptive, ce qui rend les routines plus petites, permettant donc d'avoir des temps de début (startup) et de fin (shutdown) plus rapides.

Ensuite, elles permettent d'avoir un ordre d'évaluation plus efficace, ainsi qu'un équilibrage de charge plus égal, dû à la valeur ajoutée qui découle de disposer d'un ordonnanceur de tâches spécifiques à l'application.

Puis pour finir, le niveau d'abstraction est plus élevé pour le programmeur, car il n'a pas besoin d'être compétent en matériel, c'est-à-dire de connaitre la configuration matérielle exacte de la plateforme, pour avoir une application efficace et surtout extensible à l'échelle (*scalable*).

Le type de programmation ayant été défini, nous survolerons dans les sous-sections suivantes les systèmes d'exécution communément utilisés.

2.2.2.1 Intel's Threading Building Blocks (TBB)

Tel qu'expliqué précédemment, TBB [32] est une bibliothèque d'exécution qui soutient la programmation parallèle en C++. Elle permet au programmeur de faire abstraction des fils matériels et de laisser l'ordonnanceur s'occuper de gérer l'assignation des tâches au matériel. TBB est constitué de deux bibliothèques dynamiques: une pour le support général et l'autre pour l'allocation de mémoire et la gestion des entêtes correspondantes. L'ordonnanceur de la bibliothèque dynamique se base sur la méthode du « task stealing ». Cette méthode est expliquée plus en détail dans la section 2.1.2 de ce mémoire sous le nom de l'algorithme de « Work Stealing ».

2.2.2.2 Cilk

Cilk [33], [34] est un langage de programmation multifilaire, développé par le « MIT CSAIL Supertech Research Group » en 1994. Ce langage de programmation est conçu pour explorer le parallélisme de données de façon intensive et efficace. Cilk se base sur un système d'exécution intelligent pour planifier et répartir les tâches sur les fils afin de réaliser le calcul de la manière la plus parallèle que possible. Comme TBB, Cilk utilise un algorithme d'ordonnancement se basant sur le vol de tâches (task stealing). Cilk étant une extension du C et dérivé ANSI C, il retrouve beaucoup des mots clés de ces langages. Il ajoute à ces derniers trois mots clés, *cilk*, *spawn* et *spawn* qui permettent de créer des applications parallèles et extensibles.

2.2.2.3 StarPU

StarPU [35] est un système d'exécution qui gère la planification d'un groupe de tâches sur un ensemble hétérogène d'unités de traitement, en gérant l'ordonnancement et les données de manière

portable tout en restant efficace. Il sert en général de « backend » pour la compilation de langage parallèle et les bibliothèques HPC. Les deux principales caractéristiques qui forment la base de StarPU sont tout d'abord que les tâches peuvent avoir plusieurs implémentations différentes pour un ou chacune des unités de traitement hétérogènes de la plateforme. De plus, les transferts d'ensemble de données nécessaires vers ces unités sont faits de manière transparente par StarPU.

2.2.2.4 StreamIt

StreamIt [36] , est un système comprenant un langage de streaming haute performance et un compilateur. StreamIt est construit sur un modèle synchrone avec profil de calcul continu (streaming). Ce modèle permet aux programmeurs d'implémenter des acteurs indépendants, appelé sous StreamIt comme filtres, qui interagissent sur les données des canaux d'entrée et de sortie.

La section suivante énoncera l'application pour laquelle nous voulons utiliser un système d'exécution sur une plateforme multicœurs.

2.3 Stack LTE et virtualisation d'un enodeB

La partie suivante exposera le contexte dans lequel nous voulons faire un ordonnancement de tâches sur multiprocesseur. Nous exposerons brièvement le concept de virtualisation d'une fonctionnalité sur le réseau communément appelé « *Network Function Virtualisation* » (NFV), les principaux attraits des C-RAN, et surtout comment fonctionne une pile LTE. Ainsi les spécifications du protocole LTE seront les contraintes à respecter tout le long de ce mémoire.

2.3.1 Network Function Virtualisation

Dans l'industrie des réseaux de télécommunication, la majorité des réseaux sont desservis grâce à du matériel propriétaire. Cela signifie que lorsqu'un nouveau service fait surface, ces plateformes propriétaires, qui laissent souvent peu de jeu aux utilisateurs, deviennent obsolètes et doivent être remplacées en partie pour accommoder le nouveau service. Cela augmente les coûts en énergie et en capital, c'est ici qu'entre en action le NFV.

Le NFV [9], est un concept d'architecture réseau qui vise à utiliser les plateformes de technologies informatiques propices à la virtualisation. Grâce à ce dernier, le but du NFV serait de modifier comment les opérateurs réseaux conçoivent leurs réseaux en les migrant vers des équipements de

type serveurs haut volume, commutateurs et stockage, qui pourraient être situés dans des centres de données. Ce type d'architecture rendrait les plateformes plus propices à l'évolution, pouvant donc s'adapter aux nouvelles technologies à faibles coûts, et elle serait surtout moins couteuse que lorsque les réseaux comptaient sur du matériel propriétaire. Des exemples de NFV[37] seraient la virtualisation d'équilibreurs de charge, de pare-feu, des dispositifs de détection d'intrusion et des accélérateurs sur des ressources dites « *Wide Area Network* » (WAN).

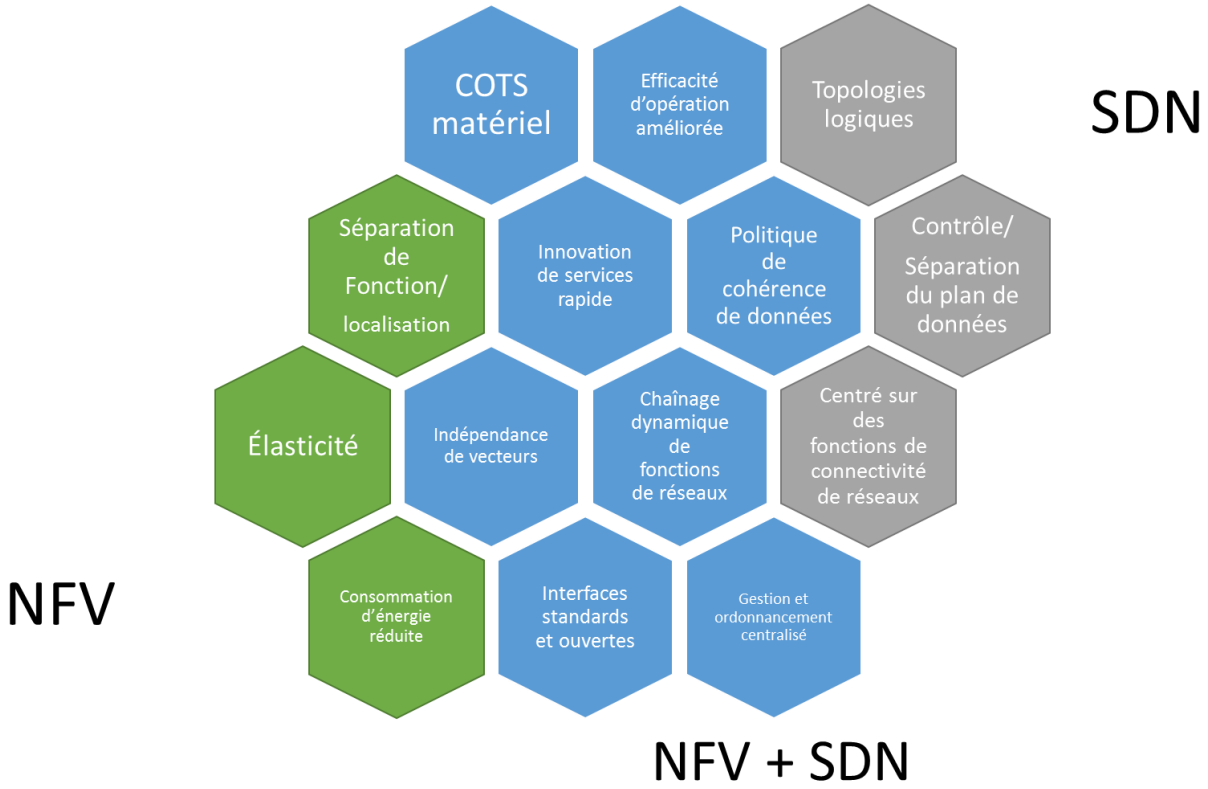

Figure 2-10 Relation entre NFV et SDN

Le NFV à une relation étroite avec le réseau défini par logiciel communément appelé « *Software Defined Networking* » (SDN). Ils sont complémentaires, mais ils diffèrent sur plusieurs points. Les buts de chacun peuvent être réalisés sans utiliser les mécanismes des autres, cependant lorsqu'ils sont utilisés ensemble, nous observons des gains en performances et cela simplifie la compatibilité avec d'autres plateformes. La figure 2-10 représente où les NFV et SDN se situent l'un vis-à-vis de l'autre. Les NFV offrent de nombreux avantages tels que :

- Réduction des coûts d'équipement et de la consommation d'énergie,
- Augmentation de la vitesse de mise en marché,
- Disponibilité de l'appareil de réseau multi-version et la multi-location

- Introduction de services ciblés pouvant être mis à l'échelle vers le haut comme le bas,
- Permettre une grande variété d'écosystèmes et encourager l'ouverture (*OpenSource*)

2.3.2 C-RAN (Cloud Radio Acces Network)

Le C-RAN [8], Cloud-RAN, ou même Centralized-RAN, est un nouveau type d'architecture de réseaux visant à améliorer les systèmes présentement utilisés dans les communications mobiles. Nous savons qu'il dérive des Radio Access Network (RAN) traditionnels, qui étaient bâtis avec de multiples stations de base (BTS) à travers une région. Une BTS (eNodeB), couvre une petite région à la fois, et le système comprend tout ce qui est nécessaire pour les communications sans-fil, du GSM 2G au LTE. La figure 2-11 expose le concept.

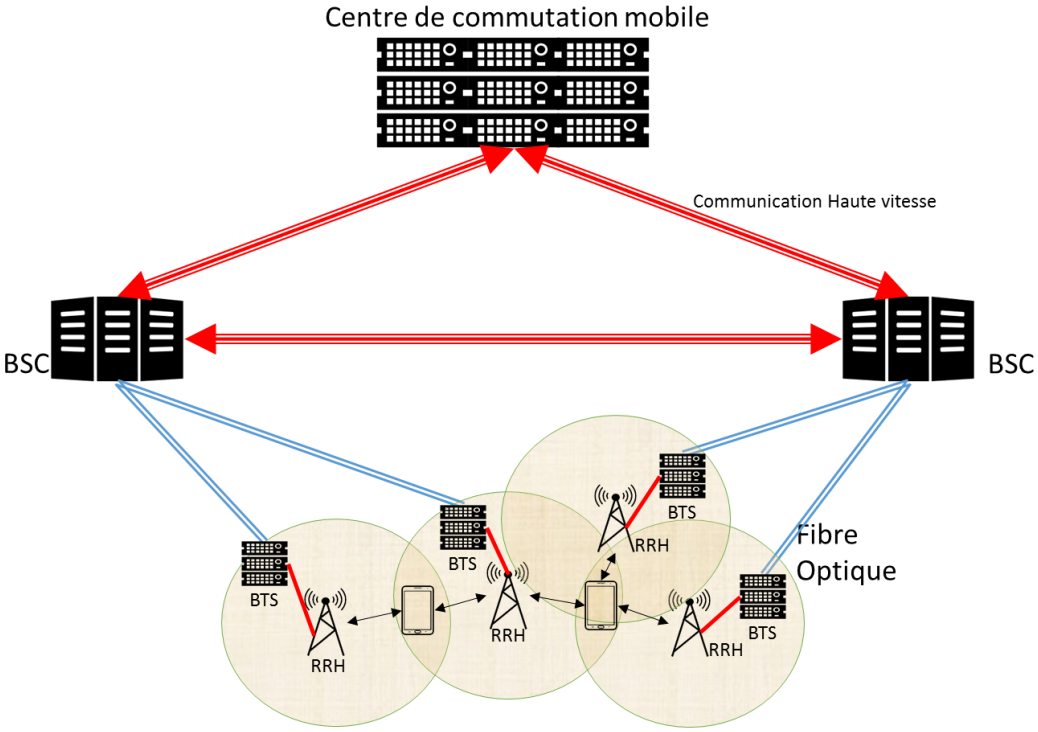

Figure 2-11 Architecture d'un Radio Access Network

Le principal désavantage qu'apporte un RAN est son coût. Tout d'abord, il y a les coûts associés au capital appelés CAPEX; les dépenses d'investissements; en effet vu que les BTS doivent avoir toutes les fonctionnalités nécessaires pour gérer des petites parties de tout un réseau, il faut donc se procurer du matériel propriétaire, avoir tous les permis, le support et les frais d'ingénierie pour concevoir le réseau. Le besoin en matériel dédié survient surtout au niveau du traitement des informations provenant et allant vers les antennes. Ce signal est normalement dans la bande de base

(*baseband*). Le besoin de matériel dédié peut aussi découler de la nécessité de supporter des fonctions de décodage et d'encodage. Des entreprises comme Alcatel Lucent[38] avec le « *Light Radio* » et Nokia Siemens Networks avec le « *Liquid Radio* », proposent du matériel dédié pour le traitement des informations en provenance des RRH. Au sein des stations de base RAN, nous retrouvons des processeurs généraux qui effectuent le reste des traitements, mais certaines fonctions sont accélérées avec du matériel, comme pour le traitement des FFT[38]. Dans le cas d'un décodeur comme FEC[39] peu parallèle, l'industrie utilise du matériel spécifique, mais notons que d'autres algorithmes de décodage, tel que le turbo, peuvent être parallélisés[39], rendant leur traitement faisable avec des multicoeurs généraux avec ou sans accélérateurs matériels.

Puis il y a les coûts d'exploitation communément OPEX (*operating expenditures*), c'est-à-dire les frais d'alimentation, de chauffage, de location ou d'achat de nombreux terrains pour pouvoir installer ces fameuses stations, et surtout mettre à jour les appareils lorsqu'un nouveau service apparait, comme l'expliquait la section précédente, ces coûts de mise à jour peuvent être important dû aux plateformes propriétaires. Ajoutons que les plateformes RAN sont peu flexibles, dû à leurs architectures. C'est-à-dire que leur demande énergétique peut difficilement être géré[40]. Prenons le cas d'une BTS à travers 24 heures, durant la journée le trafic est plus important que durant la nuit. Il est difficile de baisser la consommation d'énergie, de par le fait que désactiver une BTS au complet n'est pas une option, car nous perdrions de la couverture sur le terrain. Il faut donc une alternative en ce qui a trait aux demandes en énergie, comme une gestion dynamique des ressources au sein d'une station de base. Ainsi, plus il y a de BTS, plus les OPEX et CAPEX augmenteront. D'où la nécessité de centraliser les RAN, c'est ici que les C-RAN rentrent en jeux.

Deux architectures prévalent lorsque l'on parle de C-RAN. La version d'un réseau centralisé et d'un ensemble d'unités proches des antennes distribuées, système qui essaie de réutiliser les BTS et les convertir grâce à des NFV/SDN en un système élastique et évolutif, qui se rapporte à un nœud central. D'autre part, il y a la version complètement centralisée, qui sera l'architecture étudiée dans ce mémoire.

Le Centralized-RAN [41] , vise à rassembler les différents types de traitement de bande de base (baseband 2/3/4G/LTE) dans un même bassin de Station de Base Virtuelle (BS) afin que les ressources puissent être gérées et allouées dynamiquement en fonction de la demande sur le réseau. L'architecture centralisée est fonctionnelle grâce à trois types de composants, illustrés à la figure 2-12.

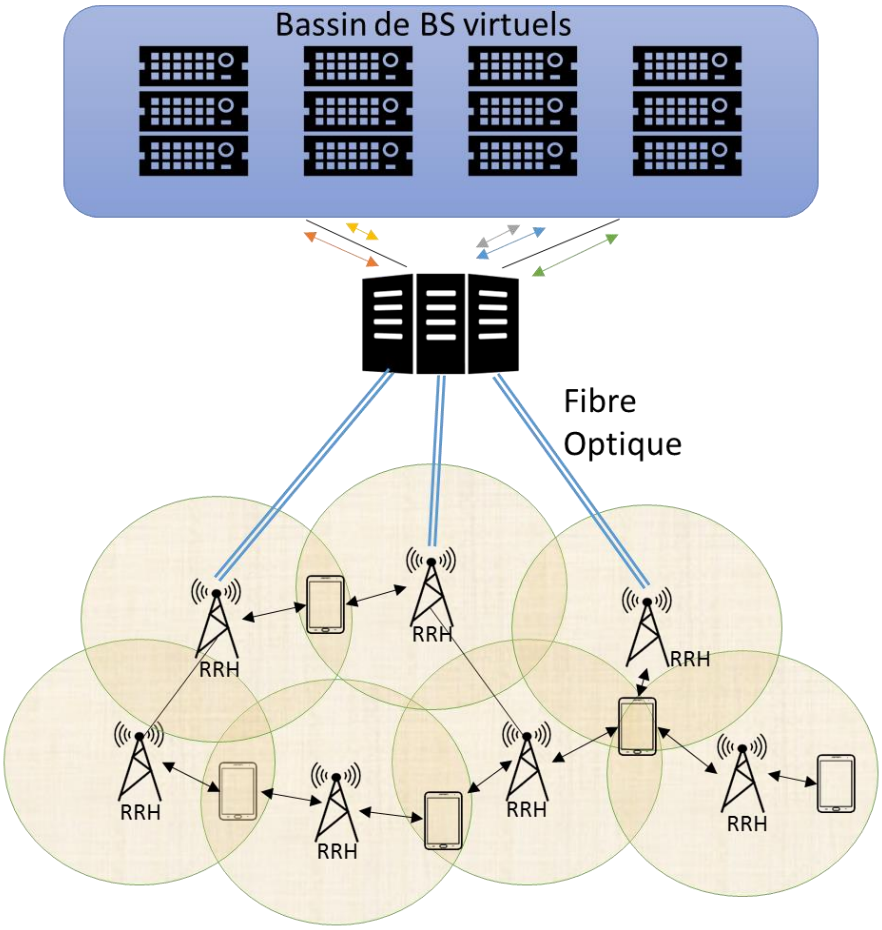

Figure 2-12 Architecture d'un C-RAN centralisée

- Le bassin d'**unités de bande de base** (Base-Band Unit, BBU), qui sera principalement formé de grappes de calculs en temps réel afin de pouvoir accommoder le réseau en ressources allouables dynamiquement tout en restant assez général pour qu'il soit évolutif, c'est-à-dire qui sera facilement mis à jour lors de l'arrivé de nouveaux services. Ce bassin se trouvera donc au sein d'un centre de données.
- Le réseau d'**unités de radio distantes (**Remote Radio Head, RRH), qui seront similaires à celles déjà sur le réseau et qui servent à fournir le réseau sans-fil de base sur toute la couverture annoncée par le fournisseur.

- Puis le réseau de **transport**, qui fournira une connexion haute vitesse, basse latence entre les BBU et RRU. Le protocole de communication peut être filaire, avec des fibres optiques par exemple, ou indirectes comme avec des microondes.

Le principal avantage du C-RAN est qu'il permettra de réduire les coûts. La centralisation des fonctionnalités sur les BBU permettra de sauver sur les CAPEX et OPEX, il sera alors plus efficace énergétiquement, donc plus vert et ultimement nous affranchira des BTS du RAN de base qui sont coûteuses. Les avantages connexes seraient de la technologie plus avant-garde qui permettrait de l'allocation de ressources dynamique, l'adaptabilité à des demandes non uniformes sur le réseau (ex : demande la journée versus le soir), des ressources de virtualisation et de« nuagification » (*cloudification*) qui donnerait l'opportunité d'utiliser des techniques et infrastructures connues du domaine des centres de données.

2.3.3 Pile de protocole LTE

Pour boucler cette revue de littérature, il faut exposer sous quelles contraintes nous voulons opérer. En vue de tout ce qui a été discuté dans des sections antécédentes, l'étude portera sur la virtualisation d'un réseau mobile conforme au protocole dits Long Term Evolution (LTE). LTE est la dernière évolution du 3GPP, et nous verrons principalement comment virtualiser la couche « PHY » du protocole. Notons que les affirmations et hypothèses énoncées se basent sur des

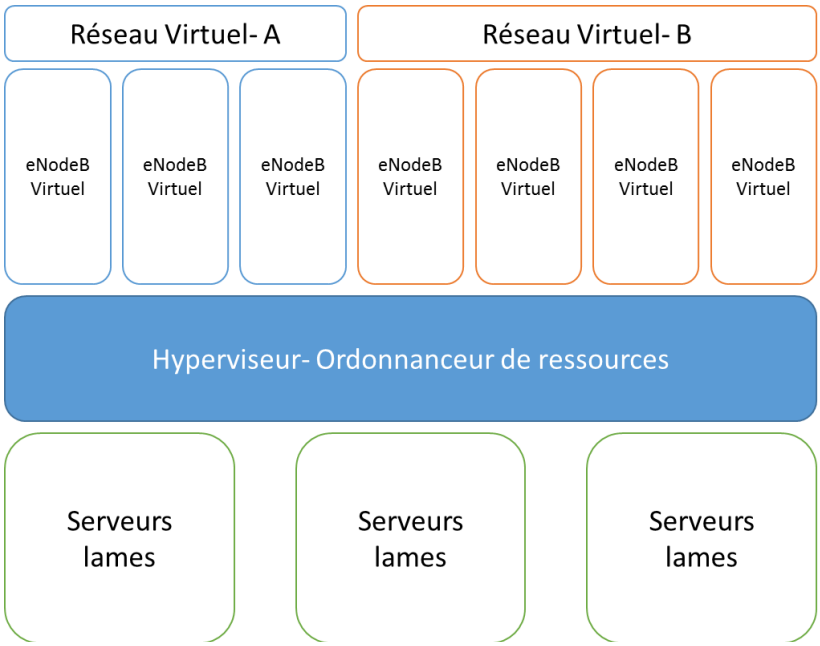

Figure 2-13 Virtualisation d'un réseau LTE

fonctions de réseaux mise en œuvre en utilisant des processeurs dans des serveurs lames *(blade servers*) en utilisant des NFV et SDN tel que l'illustre la figure 2-13 (Bassin de BBU).

Le « PHY » LTE est essentiellement full duplex [12] , il a donc une relation étroite avec les autres couches. Le protocole comporte aussi les fonctions réseaux suivantes le Media Access Control (MAC), Radio Link Control (RLC), Packet Data Convergence Protocol (PDCP), and Radio Resource Control (RRC). LTE divise le traitement en deux principales phases le « Downlink » (DL) et le « Uplink » (UL) [11]. La figure 2-14 représente la relation entre les couches du protocole par rapport aux phases.

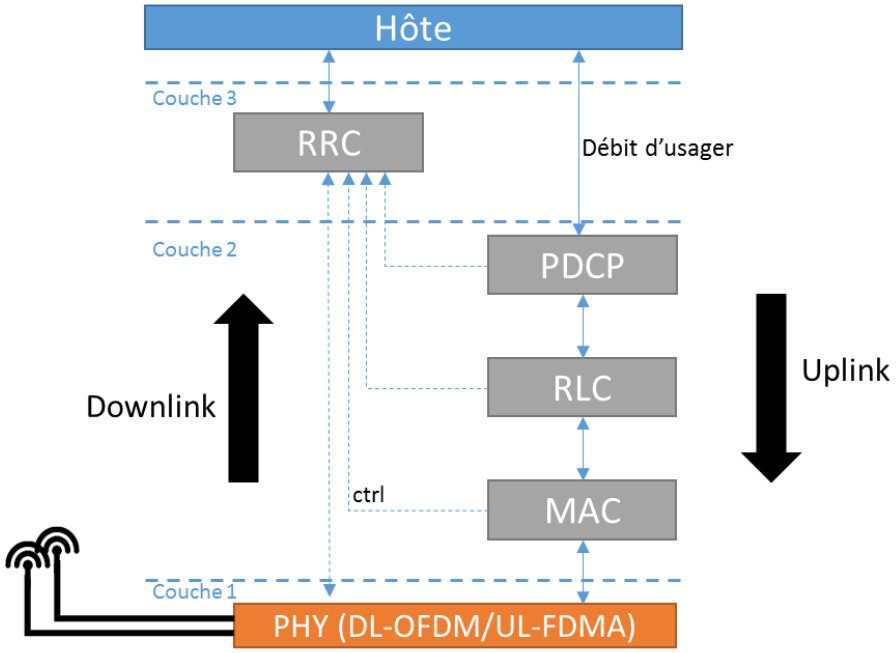

Figure 2-14 Cheminement d'un paquet LTE

Les couches PHY du *uplink* et du *downlink* ont des similitudes. En fait, elles effectuent les mêmes traitements, mais à l'inverse l'une de l'autre. Ainsi la première étape du *downlink* sur réception de données des antennes (RRH) est de prendre l'accès multiple par répartition en fréquence (FDMA) alors que la dernière étape du *uplink* est de créer l'inverse de la FDMA, une *Orthogonal frequency-division multiplexing* (OFDM). Alors le processus à travers la couche 1 est équivalent à trois étapes [1] :

- Le traitement de l'extrémité avant (front-end), qui consiste à récupérer/consolider l'information destinée à être reçue ou envoyée sur les antennes.

- Le traitement du signal par utilisateur, c'est-à-dire l'adaptation du signal en fréquence afin qu'il soit utilisable par la couche MAC ou qui le formatte en fréquence pour le renvoi vers les RRH.
- Le décodage par utilisateur, qui désentrelace et décode le signal dans le cas du Downlink, ou qui l'entrelace et encode dans le cas du Uplink.

La figure 2-15 expose les relations entre les blocs pour les deux phases du protocole.

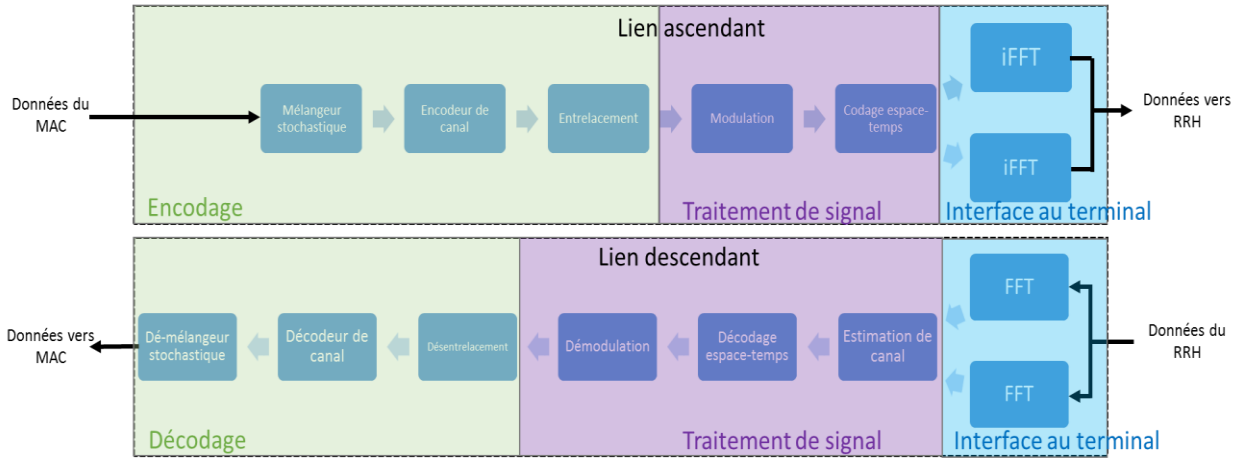

Figure 2-15 Comportement de la couche PHY au niveau algorithmique

Afin d'avoir une virtualisation fonctionnelle du protocole, il faut que le système d'exécution logiciel gère adéquatement les demandes de ressources et surtout les dépendances entre les tâches. Cependant, l'exécution séquentielle de ces tâches ne permet normalement pas de rentrer dans les contraintes de temps imposées par le protocole. Il faut donc faire appel au parallélisme pour l'exécution des tâches.

Tout d'abord, il faut définir les dépendances de blocs. Un travail préliminaire réalisé par Frigon et al[42] expose que la chaîne de dépendance la plus contraignante est celle qui existe entre les trois principaux blocs. Ajoutons le fait que des chercheurs de IBM [1] ont démontré que le protocole permet trois types de parallélisme :

- Il existe un parallélisme de bloc, pour lequel si la quantité de données le permet, on peut calculer plusieurs liens ascendants et descendants en même temps.

- Il existe aussi un parallélisme interne de lien, où chacun des blocs peut se décomposer en modules qui appartiennent à une phase et qui sont exécutables en même temps (dans la majorité des cas).
- Enfin un autre type de parallélisme est observé dans la logique de lien, lorsque les algorithmes peuvent être séparés en plusieurs sous-groupes exécutables simultanément.

Il reste donc à voir si une plateforme peut être mise en place pour virtualiser une pile LTE, le chapitre suivant développe donc une analyse théorique d'un système qui serait assez performant pour effectuer les calculs nécessaires sous les contraintes LTE.

CHAPITRE 3 ÉTUDE D'UNE GRAPPE DE CALCUL TEMPS RÉEL

À travers la revue de littérature, nous avons bien établi le contexte dans lequel nous œuvrons. Le chapitre suivant analysera une grappe de calcul adéquate pour la virtualisation d'un eNodeB pour le protocole LTE sur des processeurs à usage général (PUG). Nous verrons d'abord l'architecture du système global, puis nous évaluerons les latences et bandes passantes de chacun des protocoles inclus, pour finir avec un sommaire et une discussion sur les résultats obtenus.

3.1 Architecture du système

Dans les sous-sections suivantes, nous suivrons le cheminement d'un paquet de données, à partir de la réception sur l'antenne jusqu'à la fin de son traitement dans la grappe de calcul, c'est-à-dire jusqu'à au transfert d'information de la couche PHY à la MAC [43] . La figure 3-1 expose le système dans son ensemble.

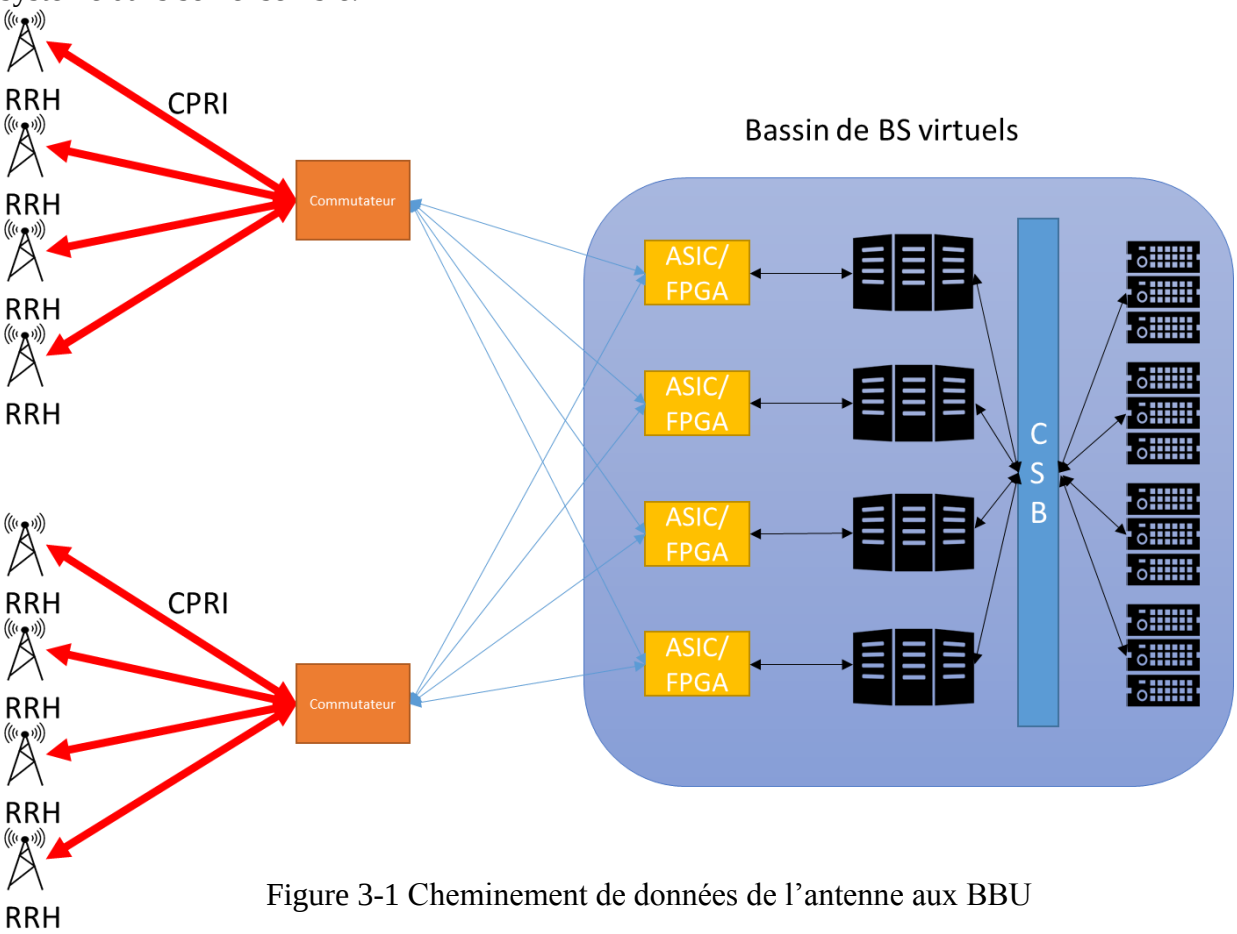

Figure 3-1 Cheminement de données de l'antenne aux BBU

Notons que les blocs commutateurs, associés au ASIC/FPGA, représentent de l'équipement radio dédié à la réception et au traitement d'un paquet CPRI. Ajoutons que ces blocs sont dédiés et qu'ils pourraient correspondre à des nœuds du BBU. Ainsi, une fois le paquet reçu sur l'équipement radio, il doit passer à travers le bassin de bases virtuelles.

3.1.1.1 Architecture de la grappe de calcul.

La grappe est formée de multiples serveurs lames, chaque serveur comporte au moins un processeur multicœurs qui servira de nœud de calcul. Un nœud sera maître et les autres seront esclaves, tous reliés par une interconnexion via un commutateur à basse latence. Le nœud maître sert de gestionnaire de ressources vers les serveurs subséquents. Les esclaves servent donc à exécuter les tâches qui leur sont fournies. La figure 3-2 illustre les faits énoncés.

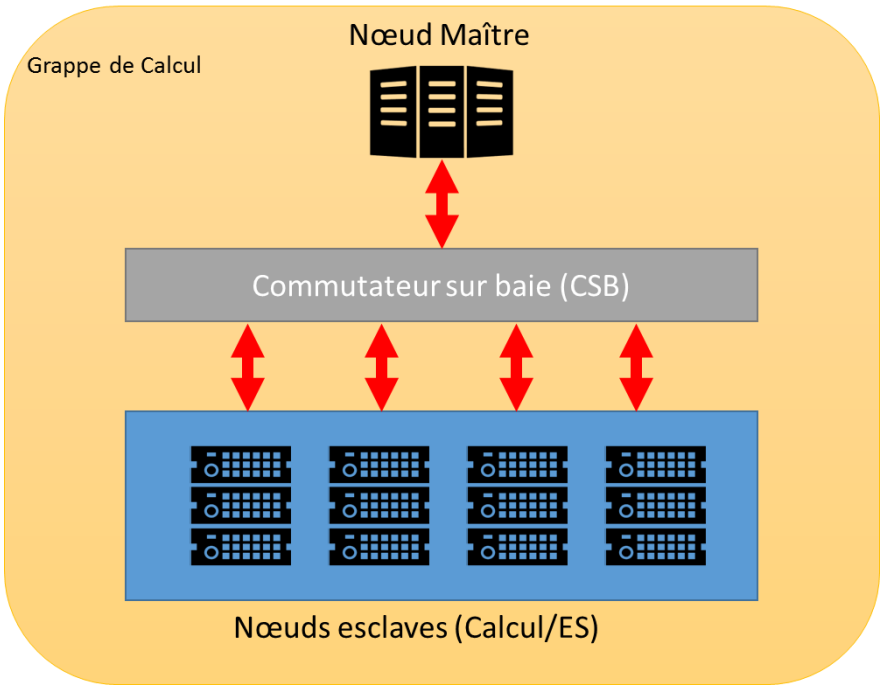

Figure 3-2 Architecture du cluster de calcul

Le maître sera une plateforme double socle hétérogène. Le premier des socles est habité par un Xeon E5 d'Intel et l'autre par un Virtex7 de Xilinx [44] . Le FPGA est donc utilisé comme un accélérateur de système d'exécution, par le biais d'accélérateur dédiés pour certains algorithmes, mais surtout par la parallélisation matérielle de l'ordonnancement des tâches à travers le système. Un autre avantage que nous voyons avec la présence du FPGA est qu'il peut aussi servir comme

accélérateur matériel de fonctions de virtualisation, tel que le Décodeur Turbo associé à une pile LTE qui est demandant en puissance de calcul pour un processeur. Puis, afin que les données puissent être transférées du maître vers les esclaves, la fente PCIe du serveur possède une VC709, qui est une carte contrôleur d'interface réseau (CIR) couplée à un FPGA. Cela permet d'avoir une implémentation personnalisée reconfigurable de l'interface de communication Ethernet. Le schéma ci-dessous expose les faits énoncés plus haut.

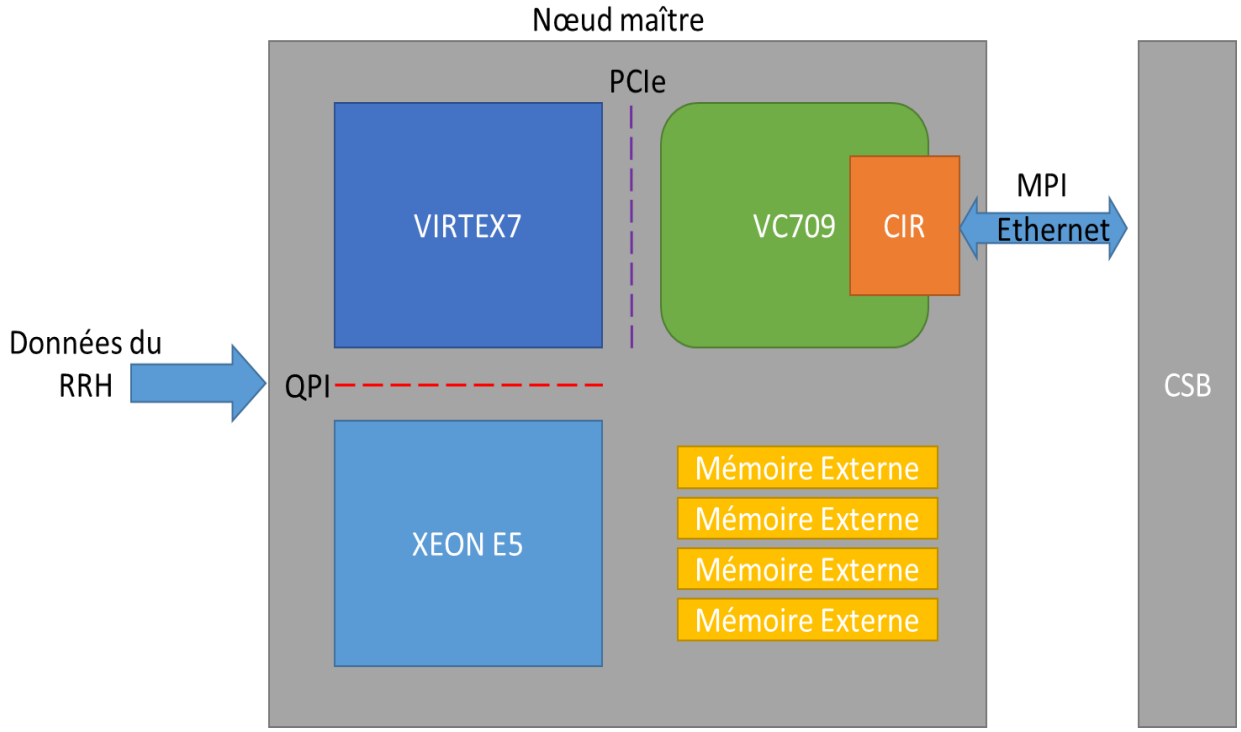

Figure 3-3 Architecture du nœud maître

En ce qui a trait aux nœuds esclaves, ils peuvent avoir de multiples configurations. Il appartient au fournisseur de décider quel type de plateformes est nécessaire. Par exemple, les serveurs lames peuvent avoir des CPU exploités comme des grappes de calcul conventionnelles, ou ils peuvent aussi être structurés comme des plateformes hétérogènes avec des GPU, des ASIC et même des FPGA.

Dans les sections suivantes, nous analyserons les communications entre chacun des blocs discutés au-dessus.

3.2 De l'antenne au bassin de stations de bases

La première communication à choisir est celle entre le RRH et le bassin virtuel. Afin de respecter les demandes du protocole LTE, deux choix s'offrent à nous, le Common Packet Radio Interface [45] (CPRI) et l'organisation proposée par l'Open Base Station Architecture Initiative RP3-01 (OBSAI RP3-01) [46] . Prenons le cas d'un lien CPRI pour notre analyse de plateforme, car il est le plus communément utilisé par l'industrie en ce moment. Nous avons donc deux principaux facteurs à prendre en considération: la latence et la bande passante.

3.2.1 Bande passante du protocole de connexion radio

Commençons par la bande passante. Nous savons que le protocole CPRI exploite une fréquence d'horloge de base à 30.72 Mhz en injectant environ 30.72 Msps. Il nous faut déterminer le débit de ligne minimal du protocole CPRI dont avons-nous besoin afin d'avoir un système MIMO 4x4. Tout d'abord nous devons déterminer combien de fréquences de transport (carriers) peuvent supporter les liens CPRI. La formule va comme suit :

$$Nbr\ de\ porteuses = \frac{nbr\_bits\_I\&Q\_bloc}{echantillion\_IQ \times taille\_echantillon \times ratio\_surechantillonage}$$

Où, *nbr_bits_I&Q_bloc* est le nombre de bits par bloc d'IQ, *echantillon_IQ* la trame de I et de Q pour chaque échantillion, *taille_echantillon* la taille de l'échantillion et *ratio_surechantillonage* correspond au nombre d'échantillons possible par échantillionnage.

Nous savons que le CPRI à quatre principaux débits binaires de ligne (line bit rate), 9.830 Gbps 6.144 Gbps, 3.072 Gbps et 2.4576 Gbps. Prenons un débit de 2.4576 Gbps, associé à un échantillonnage de 16 ou 15 bits et un bloc IQ de 480 bits. D'après le document de spécification du protocole CPRI [45] , nous savons que la période (T_c) que prend un trame de base est de :

$$T_c = \frac{1}{f_c} = \frac{1}{3.84\ MHz} = 260.4\ ns$$

Cela nous permet donc d'estimer notre ratio d'échantillonnage dans le contexte LTE à 8. Ainsi, grâce à toutes ces informations nous pouvons en déduire que le nombre de porteuses possibles sur une ligne CPRI à 2.4576 Gbps est :

$$Nbr\ de\ transporteurs = \frac{480}{2 \times 15 \times 8} = 2$$

Nous pouvons donc gérer deux porteuses sous les spécifications ci-dessus. Si nous faisons la supposition d'une porteuse par antenne, la configuration considérée permet 1 RRH avec une configuration MIMO 2x2 pour un protocole LTE à 20 MHz. Ainsi, nous calculons les configurations possibles dans le cas des différents débits de ligne de CPRI. Le tableau 3-1 expose les résultats obtenus :

Tableau 3-1 Capacités du système pour différents débits de ligne CPRI

Débit de ligne (Gbps)	Configuration des MIMO	Canal de 20 MHz
2.4576	2x2	1 RRH
3.072	2x2	1 RRH
6.144	2x2	2 RRH
	4x4	1 RRH
9.830	2x2	4 RRH
	4x4	2 RRH
	8x8	1 RRH

Nous avons démontré qu'un débit de ligne de 2.4576 Gbps peut gérer deux porteuses, pour une configuration MIMO 2x2 pour 1 RRH. Alors afin que nous puissions avoir notre configuration MIMO 4x4 sur un RRH, il faudrait un débit de ligne deux fois plus élevé que le précédent, le système aura donc besoin d'une ligne CPRI d'au moins 6.144 Gbps.

3.2.2 Latence du protocole et de traitement

Nous avons déjà défini que le protocole à une latence de base de 260.4 ns dû à l'horloge de base de CPRI. Ainsi, nous devons essayer de définir la latence associée au traitement des informations transportées par une ligne CPRI vers la plateforme.

Cependant, cet aspect est entièrement dépendant du système utilisé. Deux options s'offrent à nous, utiliser un ASIC dont la principale tâche serait de gérer le traitement de trames CPRI sous des spécifications précises ou utiliser un FPGA, qui rend le contrôle accru et permet surtout une adaptation au travers les augmentations de performance des protocoles radio. À travers la littérature nous observons que nous pouvons obtenir des performances équivalentes à une latence de 1.5 µs grâce à des IP sur FPGA comme le Prism IQ [47] . Nous pouvons aussi améliorer ce genre de performance par le biais de compression 2 :1 de la trame CPRI.

3.3 Communication CPU-FPGA

Cette sous-section est particulière, car nous aurons une analyse théorique, suivie d'une analyse de latence effectuée sur le système en question.

3.3.1 Analyse théorique

Pour la communication entre un CPU et un FPGA, deux options prévalent: le Quick Path Interconnect [48] (QPI) d'Intel et Peripheral Component Interconnect Express [49] (PCIe). Le protocole de communication affecte grandement l'architecture du système. QPI possède 20 lignes full duplex, qui par le biais du Xeon E5-2670 offre une bande passante de 8 GT/s. Il nous permet donc d'utiliser une architecture double socle, où l'on populе le premier avec le Xeon et l'autre avec le FPGA. La théorie veut qu'avec ce type d'architecture, la communication CPU<->CPU devrait être similaire pour ne pas dire la même que CPU<->FPGA. Cela signifie qu'avec ce type de plateforme, nous pouvons avoir une faible latence avec une haute bande passante, nous permettant donc d'avoir une accélération sur des tâches de plus basse granularité. Le principal désavantage du QPI est qu'il est un bus propriétaire, contrairement au PCIe qui est ouvert et très utilisé dans l'industrie.

Comparons donc les protocoles de communication. Grâce au Xeon E5 nous pouvons atteindre 8 GT/s et le PCIe Gen3 x8 nous donne la même capacité de transfert. Cependant, QPI est un protocole complètement dépendant de son hôte. Le calcul de sa bande passante va donc comme suit :

$$QPI\ bandwidth = \frac{freq \times data\ rate \times \frac{bits}{liens\ QPI} \times duplex}{nbre\ de\ bits\ par\ octets}$$

La valeur absolue dépend grandement du système de test, mais suivant la formule ci-dessus notre bande passante de QPI devrait être :

$$QPI\ bandwidth = \frac{2.6\ GHz \times 2 \times 16 \times 2}{8} = 20.8\ GB/s$$

Ainsi en comparaison avec le PCIe nous avons :

Tableau 3-2 Bandes passantes calculées de QPI et PCIe Gen 3

	QPI	PCIe Gen 3
Bande Passante	20.8 GB/s	x8 = 7.88 GB/s [49]
Type	Propriétaire	Ouvert
Direction	Double	Simple

Notons que le QPI est une communication basée sur des transferts par l'antémémoire. Ce protocole agit par la modification et invalidation de lignes d'antémémoire communes entre processeurs. Cela signifie que les transferts sont cohérents en antémémoire, ce qui ne nécessite donc pas de latence ajoutée due à la gestion des interruptions contrairement au PCIe, qui offre en général une latence deux fois plus élevée que le QPI [50], [51].

Nous avons donc choisi le QPI pour la plateforme ciblée dans ce mémoire. Dans la prochaine sous-section, nous effectuerons donc une brève analyse de latence sur le système en question.

3.3.2 Analyse empirique

Commençons par définir l'environnement de test. L'architecture globale du système correspond à la figure 3-3 plus haut. Il s'agit d'une architecture double socle, avec un Xeon E5-2670 dans le premier et un Virtex 7 dans le second. Nous avons donc une connexion filaire entre le CPU et FPGA par QPI. Le but des tests de la section suivante sera de confirmer la faisabilité d'une communication d'antémémoire cohérente, en plus de voir quelles latences et bandes passantes nous pouvons obtenir empiriquement.

3.3.2.1 Environnement de travail

Si nous rentrons plus dans le détail, nous réalisons des tests avec un environnement tel que présenté par le schéma bloc ci-dessous :

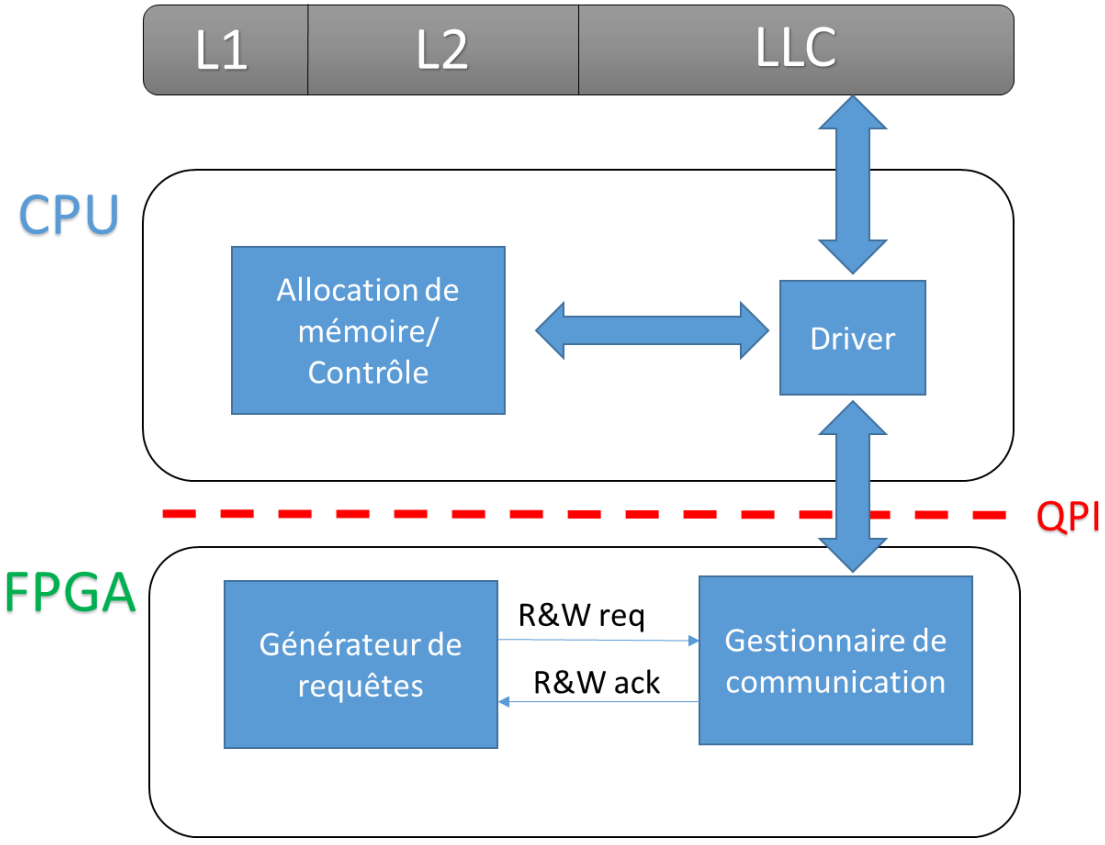

Figure 3-4 Schéma bloc du test de latence et de bande passante du port QPI

Nous avons quatre principaux blocs :

- L'**application**, qui roulera sur un système Ubuntu 14.04. Elle servira de contrôle pour le test, c'est-à-dire pour gérer les registres et informations à envoyer vers le FPGA. De plus, cette application effectuera l'allocation de mémoire par le biais de la biblique DPDK, qui nous permet de gérer des « *hugepages* ». Une *hugepage* est un espace de mémoire virtuelle de 2 Mo pour une architecture x86 (i7 et Xeon E5).
- Le **pilote**, fourni par Intel et Xilinx, nous l'utilisons pour établir la correspondance entre les adresses virtuelles et les adresses physiques spécifiques à la machine. Ajoutons que bien que le pilote a été fourni, nous avons dû le modifier pour qu'il soit fonctionnel pour nos tests. Nous avons dû modifier la configuration des accès aux adresses virtuelles et physiques, afin d'avoir la configuration adéquate pour nos tests. Combiné avec des ajouts au niveau des fonctions allocation et libération d'espaces mémoire.
- Le **gestionnaire de communication,** aussi appelé Cache Coherent Interface (CCI) permet de gérer les horloges et bus de transmission de la communication QPI. Ce dernier est un IP de Xilinx, qu'on peut retrouver sur leur site sous le nom de qpi_core [52].
- Puis le **générateur de requêtes**, qui comporte nos tests implémentés en Verilog. Ces derniers envoient des requêtes de lecture et/ou d'écriture vers le CPU et attendent une confirmation par la suite.

3.3.2.2 Méthode de test

Les sous parties ci-dessous exposeront les différentes méthodes utilisées pour les tests de latence et de bande passante de l'interface QPI.

3.3.2.2.1 Test de latence

Grâce à ce système nos effectuerons les tests de bande passante et de latence du port QPI. Tous les tests sont faits avec des données de 64 octets, ce qui équivaut à une ligne d'antémémoire.

Le test de latence va comme suit :

1) Initialisation des *hugespages* et du pilote.
2) Sélection du test à faire.

3) Configuration des registres de contrôle.
4) Commencer le test envoyé par le CPU.
5) Lancer un compteur en roue libre.
6) Faire une écriture d'une ligne d'antémémoire vers l'antémémoire du CPU.
7) Prendre la valeur du compteur lors de la requête.
8) Attente et Réception de l'ack du CPU.
9) Récupération de la valeur du compteur.
10) Flag de test terminé mis à 1.
11) L'application vérifie la valeur lue à l'adresse X, si elle est bonne le test est valide.

Le test de latence d'écriture est fait de manière similaire, mais avec une requête d'écriture. Le test peut-être fait en deux parties distinctes, écriture puis lecture et vice versa, ou faire une copie dans l'antémémoire, donc lecture d'une adresse puis écriture des données lues à l'adresse +1.

Lors d'une écriture ou lecture, deux choses peuvent se passer, avoir une présence dans l'antémémoire ou une absence dans l'antémémoire. Afin de faciliter la compréhension de ces

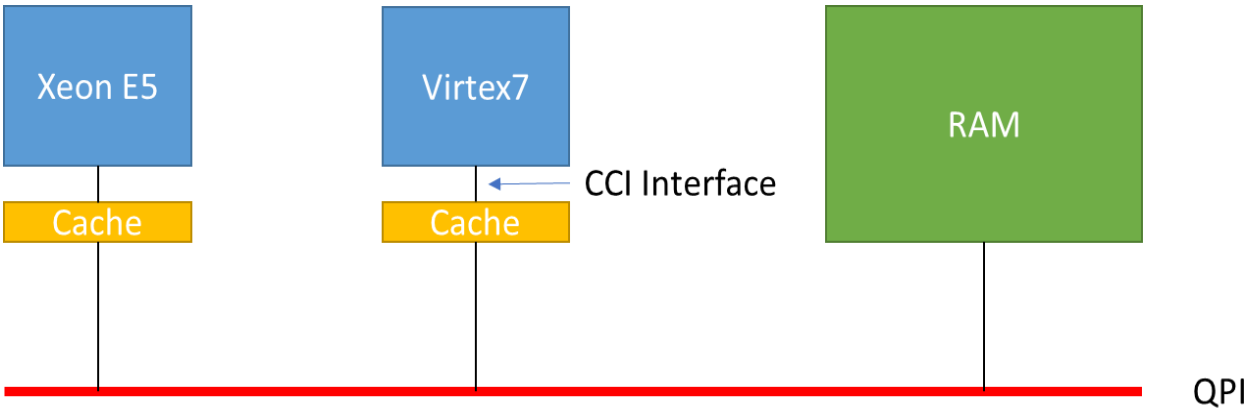

Figure 3-5 Relation entre le CCI, le FPGA et le CPU

phénomènes, la figure 3-5 suivante expose l'interconnexion entre le CPU et FPGA. En effet, l'interface CCI est entre le FPGA et son antémémoire alors que le QPI est entre l'antémémoire du FPGA et le CPU. Ainsi, dans la plupart des cas, lorsqu'une présence en antémémoire survient, cela signifie que l'écriture ou la lecture a été faite dans l'antémémoire du FPGA en premier ou dernier lieu. Cette latence sera donc la plus courte. Alors dans le cas contraire, une absence en antémémoire signifie que le système saute l'antémémoire du FPGA pour aller directement vers le CPU.

Examinons donc les différents scénarios d'écriture et de lecture en antémémoire au sein du système. Si nous cheminons dans le même ordre que la méthode de test, nous commençons par la configuration des registres de contrôle du FPGA initiés par le CPU. Le diagramme de séquence ci-dessous montre qu'une fois la requête envoyée par le CPU, le FPGA renvoie automatiquement une note de complétion.

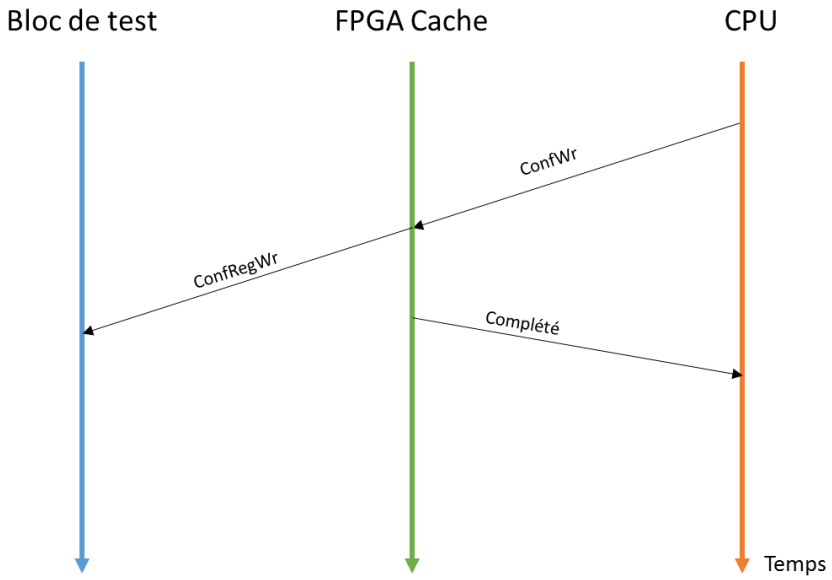

Figure 3-6 Configuration des registres de contrôles du CPU

Ensuite, il y a l'envoi de la requête d'écriture du Virtex vers le Xeon. Si une présence dans l'antémémoire survient, le FPGA renverra directement une réponse vers le bloc de test. Nous restons donc au sein de l'interface CCI sans jamais accéder au bus QPI, comme le montre la figure 3-7.

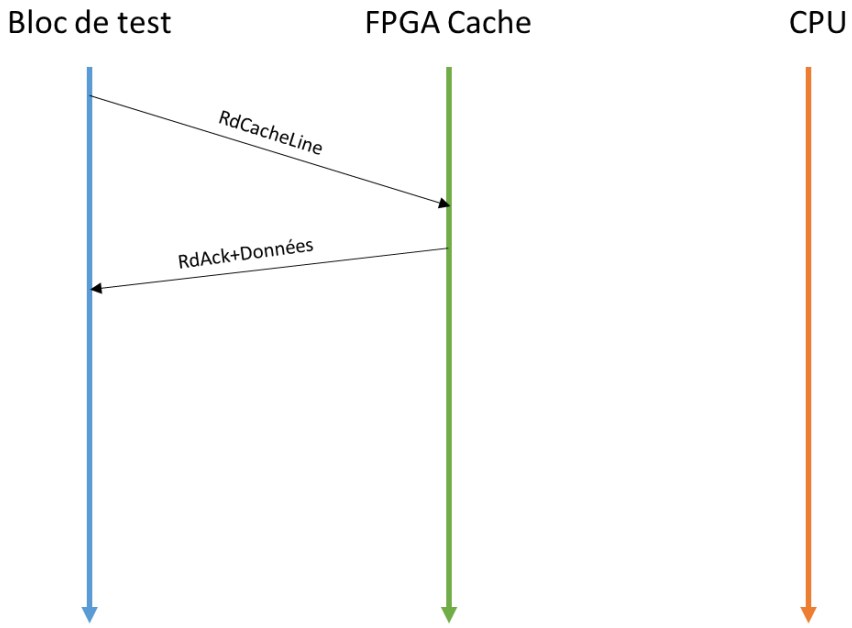

Figure 3-7 Lecture avec présence dans l'antémémoire

Dans le cas d'une absence dans l'antémémoire, une requête de lecture sera acheminée vers le CPU par l'interface QPI. Une fois la requête acceptée, une réponse de complétion est renvoyée vers le FPGA tout en marquant la ligne d'antémémoire lue à l'état partagée, la figure 3-8 expose ce fait.

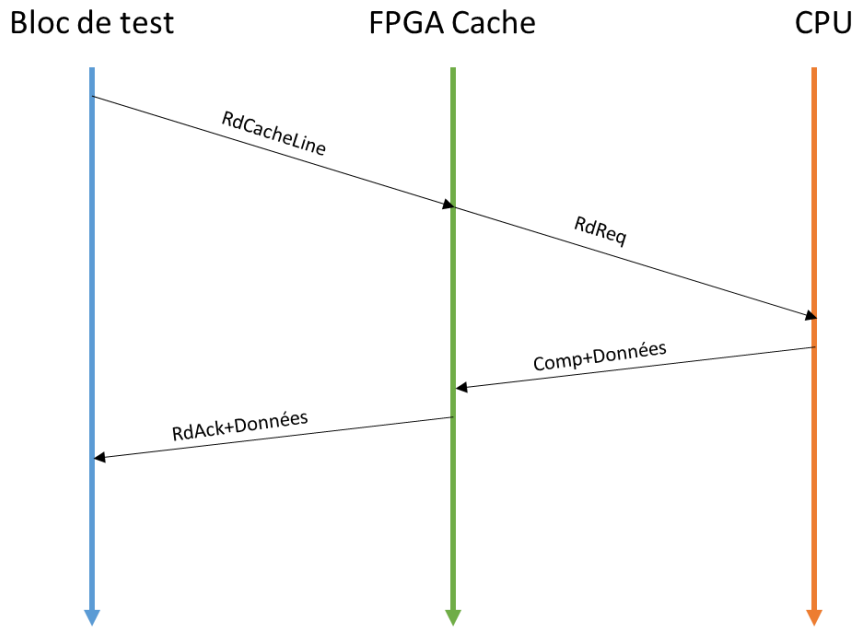

Figure 3-8 Lecture avec absence dans l'antémémoire

Avant d'expliquer le comportement d'une demande d'écriture, nous devons expliquer les deux différents modes d'écriture, nous avons :

- L'**écriture directe** (WriteLine), qui est préférable lorsque le bloc de test ou les accélérateurs sur le FPGA ont besoin d'accès multiples à cette même ligne d'antémémoire. Ainsi, si la ligne vise à être réutilisée, il est préférable de faire une écriture directe.
- L'**écriture immédiate** (WriteThru), qui vise à maximiser la bande passante du port QPI en optimisant les flux sur le port de manière balancée. Cependant, l'écriture immédiate crée beaucoup d'écritures différées (Write-Back) et d'expulsion pouvant amener à des pertes d'information si le lien QPI est utilisé proche ou au maximum de sa capacité.

Prenons le cas d'une écriture directe, qui effectue une présence dans l'antémémoire. Le FPGA initie la requête d'écriture, et lors de la réception au niveau de l'antémémoire du FPGA, elle renvoie directement une réponse de complétion. La figure 3-9, expose le phénomène.

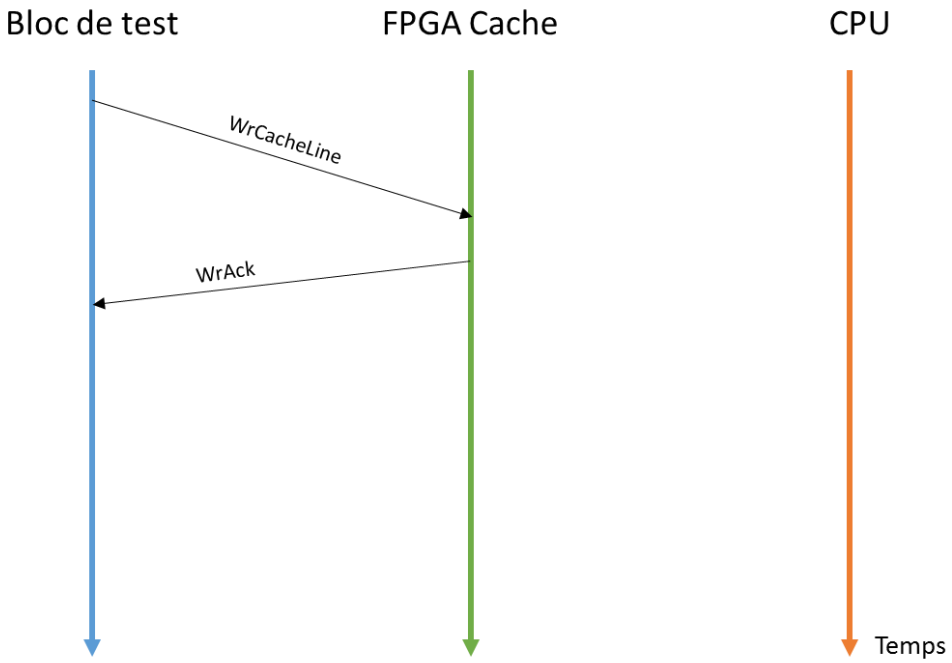

Figure 3-9 Écriture directe avec présence dans l'antémémoire

Dans l'éventualité d'une absence dans l'antémémoire lors de l'écriture directe, une demande d'écriture est acheminée vers le CPU en passant par le QPI, lors de sa réception une réponse de complétion est renvoyée vers le bloc de test, comme vu sur le diagramme ci-dessous.

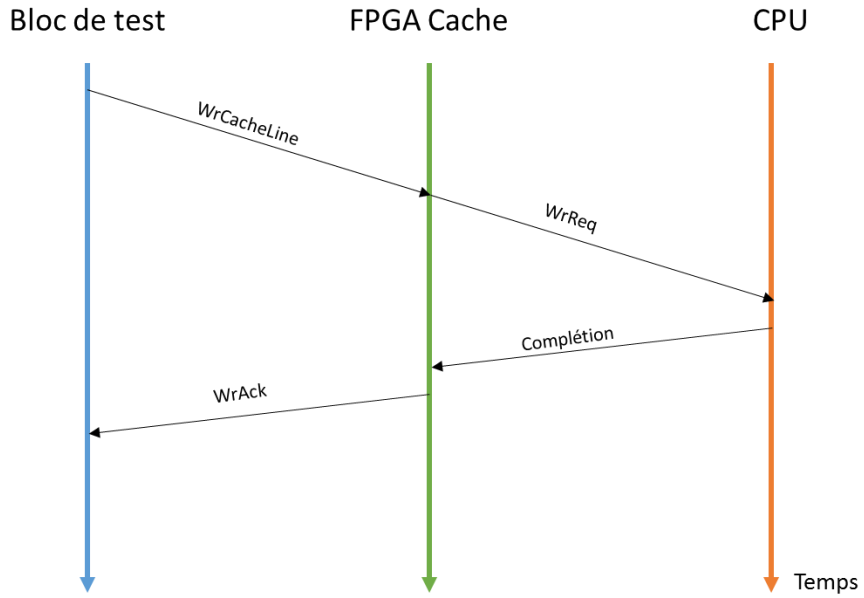

Figure 3-10 Écriture immédiate avec absence dans l'antémémoire

Passons à l'écriture immédiate avec une présence dans l'antémémoire, le bloc de test envoie la requête vers le CPU, cependant la réponse est directement renvoyée par le FPGA. Puis une fois

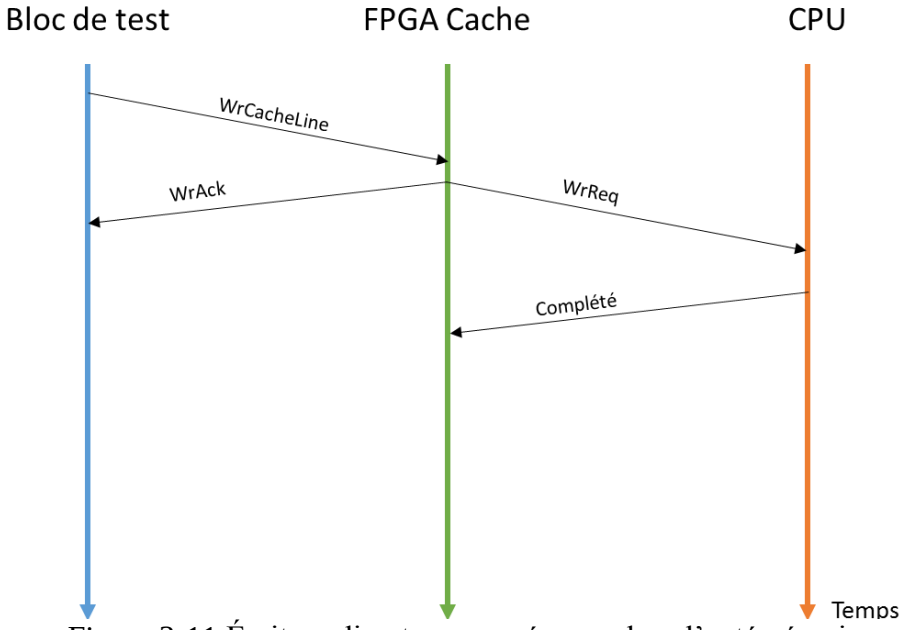

Figure 3-11 Écriture directe avec présence dans l'antémémoire

que la requête est reçue par le CPU, une réponse de complétion est dirigée vers l'antémémoire du FPGA, comme le montre la figure 3-10.

Pour finir, nous avons le cas de l'écriture immédiate avec une absence dans l'antémémoire. Une requête d'écriture est faite aux CPU par l'interface QPI, une fois reçue une réponse d'autorisation retour vers le FPGA qui lui envoie une réponse de complétion vers le bloc de test. Pendant ce temps-là, la ligne d'antémémoire de FPGA est mise invalide, l'écriture se fait au niveau du CPU puis elle finit par renvoyer une réponse de complétion. Le diagramme 3-12 illustre cette situation.

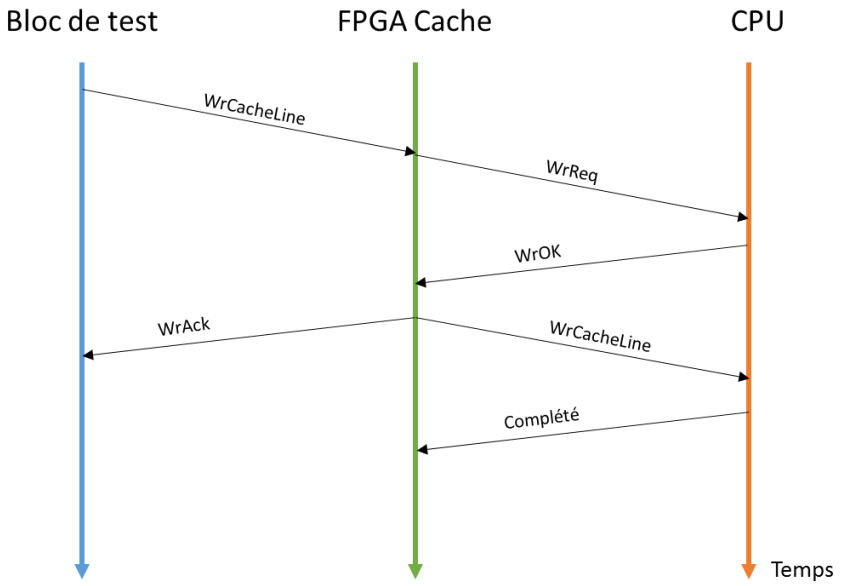

Figure 3-72 Écriture immédiate avec absence dans l'antémémoire

3.3.2.2.2 Test de bande passante

Comme expliqué dans la sous-section précédente du chapitre, le comportement d'antémémoire reste le même, mais le test change. Pour effectuer un test de bande passante, nous voulons maximiser l'utilisation de l'interface QPI afin de pouvoir lire et écrire le plus grand nombre de données possibles. Ainsi, le bloc de test initiera le plus grand nombre de requêtes possibles sans vérification du mot lu ou écrit. Le test va donc comme suit :

1) Initialisation des *hugespages* et du pilote.
2) Sélection du test à faire.
3) Configuration des registres de contrôle

4) Commencer le test envoyé par le CPU.
5) Lance un compteur de roue libre
6) Faire des requêtes d'écriture de multiples lignes d'antémémoire vers le CPU.
7) Prendre la valeur du compteur lors de la première requête.
8) Récupérer les nombres de requêtes envoyées.
9) Récupérer les nombres de réponses reçus.
10) Prendre la valeur du compteur lors de la dernière réponse.
11) Flag de test terminé mis à 1.
12) L'application vérifie la valeur écrite à l'adresse X, si elle est bonne, le test est valide.

Le test de bande passante lors de l'écriture est fait de manière similaire, mais avec des requêtes d'écriture. Nous exposerons les résultats obtenus lors des tests dans la section suivante.

3.3.2.3 Résultats

Afin que les résultats soient compréhensibles, nous devons clarifier les faits suivants.

- La cadence de l'horloge du FPGA est de 200 MHz, dû à la complexité du qpi_core d'Intel. Cela signifie qu'un cycle d'horloge équivaut à 5ns.
- Autre fait important, la vitesse de transmission sur le port QPI est limitée à 6.4GT/s au lieu du 8 GT/s (capacité du Xeon E5-2670) dû à la version du qpi_core utilisé. Le Virtex 7 modèle XC7VX485T possède 56 transducteurs GTX qui permettent une vitesse de transmission de 12.5Gb/s [53] chaque. Nous avons besoin d'utiliser 20+1 transducteurs afin de pouvoir utiliser le module IP gérant la partie PHY de la communication.

Notons que la vitesse de transfert pourra être augmentée lors de l'arrivée sur le marché des nouvelles puces de Xilinx, Virtex Ultrascale [54] qui permettront jusqu'à 16.3Gb/s pour les GTH et 30.5Gb/s pour les GTY.

Les ressources FPGA utilisées pour faire les tests sont les suivantes :

Tableau 3-3 Ressources FPGA utilisées pour les tests de l'interface QPI

	LUT	FF	BRAM	GT
QPI core	40 575 (8.3%)	55 300 (9.1%)	185 (18%)	21 (37.5%)
Bloc Test	3 300 (0.7%)	3 100 (0.5%)	7 (0.7%)	0
Total	43 875 (9%)	58 400(9.6%)	192 (18.7%)	21 (37.5%)

Nous observons que la majorité des ressources utilisées sont au niveau des entrées /sorties (Transducteurs). Nous utilisons 37.5% des GT, il reste donc 62.5 % d'entre eux si nous voulons utiliser d'autres protocoles de communications en addition du QPI, tel que PCIe pour d'autres accélérateurs de fonctions. Les tableaux ci-dessous exposent les résultats obtenus lors des tests empiriques. Les tableaux 3-4 et 3-5 sont des moyennes des valeurs obtenus après de nombreux tests de latence et de bande passante.

Tableau 3-4 Résultats des tests de latence de l'interface QPI

	Lecture échec	Lecture succès	WriteLine échec	WriteLine succès	WriteThru échec	WriteThru succès
Latence (cycles du FPGA)	117	15	115	25	125	105

Tableau 3-5 Résultats des tests de bande passante de l'interface QPI

	Lecture	Écriture
Bande passante (GB/s)	6.3	6.003

Contenue des résultats obtenus ci-dessus nous venons à la conclusion que QPI est adéquat pour notre application. Nous voulions avoir une communication d'une latence inférieure à celle offerte par PCIe tout en obtenant une bande passante plus élevée. Nous observons que nous obtenons la bande passante spécifiée, mais surtout que la latence associée à la lecture ou à l'écriture dans l'antémémoire est très basse, 75 ns dans le cas d'une lecture. Ajoutons aussi que l'arrivée de nouvelles technologies de FPGA couplées avec les nouveaux processeurs d'Intel qui offrent une vitesse de transferts QPI à 9.6GT/s, nous permettra une communication plus rapide et plus fiable. De plus, la latence ainsi que la bande passante peuvent être améliorées en révisant l'implémentation du bloc test et de l'interface CCI. Pour le cas où nous obtenons la plus basse latence possible avec une plus grande bande passante l'architecture de transfert de données peut-être simplifiée, mais nous sacrifierons de l'intégrité de données pour le faire. De plus si nous voulons avoir une bande passante plus élevée, nous pouvons aussi implémenter une deuxième interface QPI, le Xeon [55] nous offre deux liens avec lesquels communiquer et le Virtex7 nous permet l'implémentation d'une deuxième interface, dû à la faible demande de ressources logiques et de mémoire du protocole.

3.4 Communication FPGA vers CIR

Comme le montre la figure 3-13, nous voulons utiliser une VC709 comme CIR. Cette carte comporte un Virtex7 XC7VX690T-2FFG1761C [56] et se connecte par PCIe dans un port de la carte mère. Normalement, nous voudrions communiquer avec cette carte par le protocole PCIe gen3. Cependant, grâce au fait que nous avons une connexion FPGA à FPGA, nous pourrions améliorer la latence et la bande passante de cette interface. Pour ce faire, il faut analyser trois aspects du serveur, l'architecture d'une carte mère double socle et l'architecture des connexions vers le socle nécessaire.

Tout d'abord, nous voulons créer une communication plus efficace en utilisant le matériel dédié à la communication PCIe. Dû à l'architecture de notre système, nous avons une connexion filaire d'un FPGA à l'autre. En effet, à partir du VC709, la puce est connectée aux broches du PCIe, qui sont connectées à un port directement relié au circuit de support du socle du processeur ou directement au socle lui-même dépendant de la structure de la carte mère. Comme l'expose le schéma 3-14, le circuit de support nommé « *Chipset* » gère les connecteurs et communications

disponibles sur une carte mère en relation avec le CPU. Afin de pouvoir faire une communication ad hoc nous voulons donc une connexion telle qu'illustrée par le schéma ci-dessous.

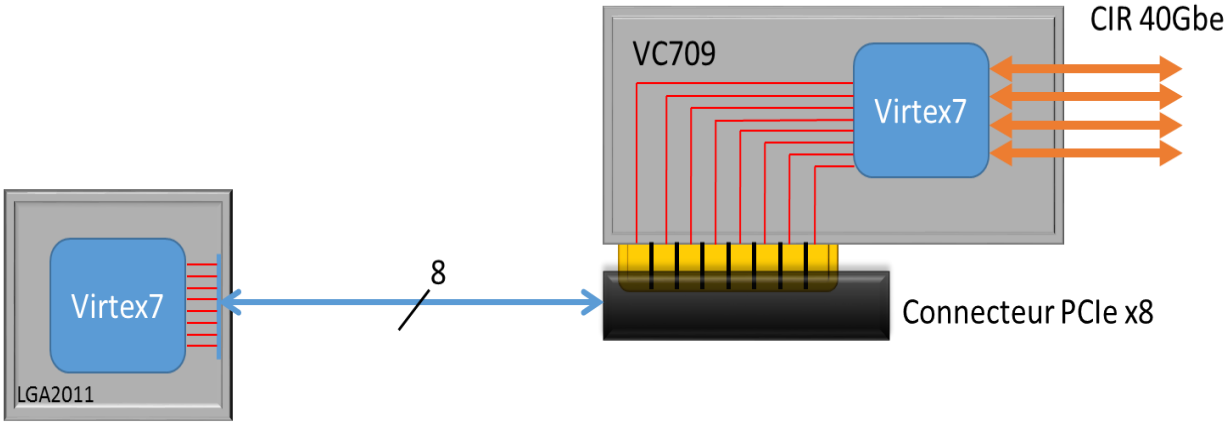

Figure 3-13 Connexion filaire entre le VC709 et le Virtex 7 « in-socket »

Afin d'avoir ce type d'architecture, il faut s'assurer que le socle et la carte mère ont un accès direct vers le port PCIe. Pour que le Virtex 7 soit compatible avec le socle, nous devons avoir un socle de type LGA2011 et la carte mère [57] doit avoir l'architecture suivante :

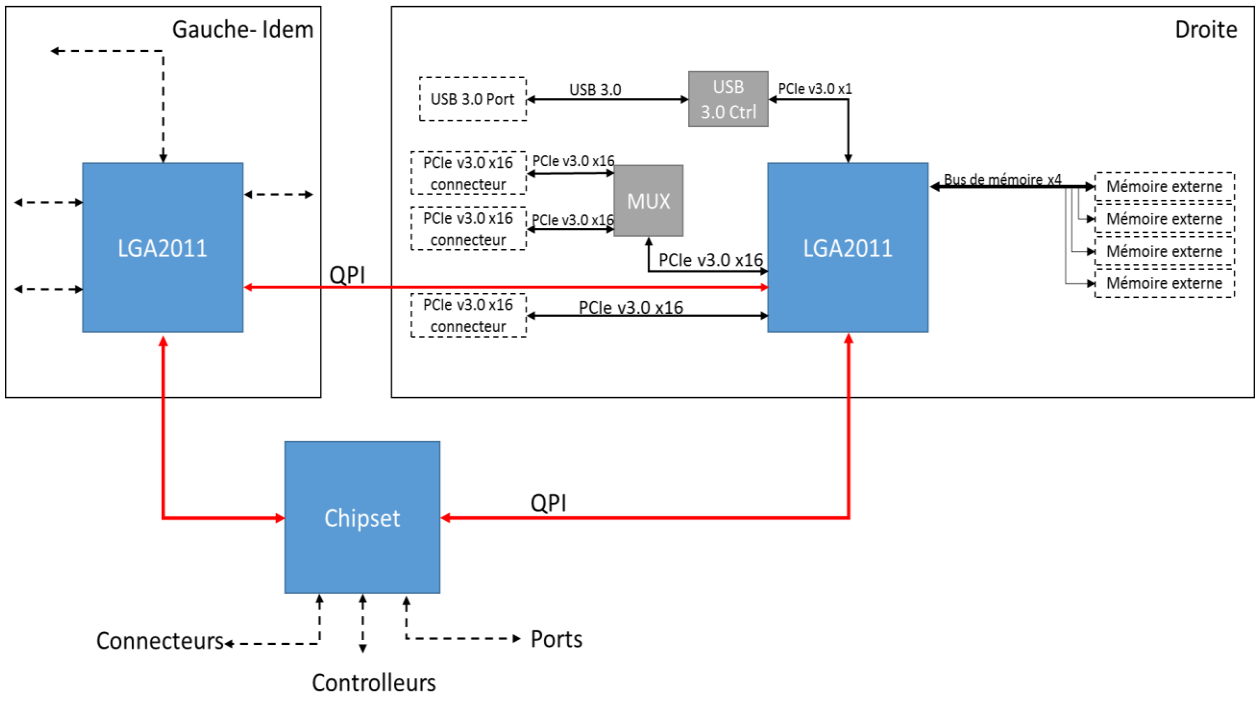

Figure 3-14 Architecture d'une carte mère double socle

Ensuite, nous devons comprendre comment est constituée la couche PHY du protocole PCIe[49] . Le VC709 comporte un lien x8 troisième génération, cela signifie que nous aurons besoin de 8 paires différentielles pour le transfert de données.

Puis, afin de pouvoir créer notre communication ad hoc, nous devons voir si les ressources nous le permettent. Dans la partie précédente, nous avons vu qu'il reste 35 transducteurs libres pour la communication. Nous avons besoin des deux côtés d'avoir 8 transducteurs afin de pouvoir faire l'implémentation de notre système ad hoc. Du côté du Virtex embarqués, nous devrons utiliser 8 transducteurs GTX 12.5Gb/s, alors que sur la VC709, nous devrons utiliser des GTH à 13.1Gb/s. La bande passante sera donc limitée par les GTX. Le tableau 3-6 expose donc la bande passante théorique que nous pourrions avoir.

Tableau 3-6 Bande passante d'une communication ad hoc versus PCIe

	communication Ad Hoc	**PCIe Gen 3 x8**
Bande passante	12.5GB/s	7.88 GB/s

Si nous voulons optimiser le protocole pour avoir la plus basse latence possible, nous devons créer un circuit de contrôle minimal. Il devra contenir un bloc de phasage afin de réduire le jigue et les biais de synchronisation, un pour la polarité du signal, un bloc d'encodage, une FIFO d'ajustement avant de passer à travers les pilotes de la carte. Le schéma bloc suivant expose la couche PHY d'une interface possible.

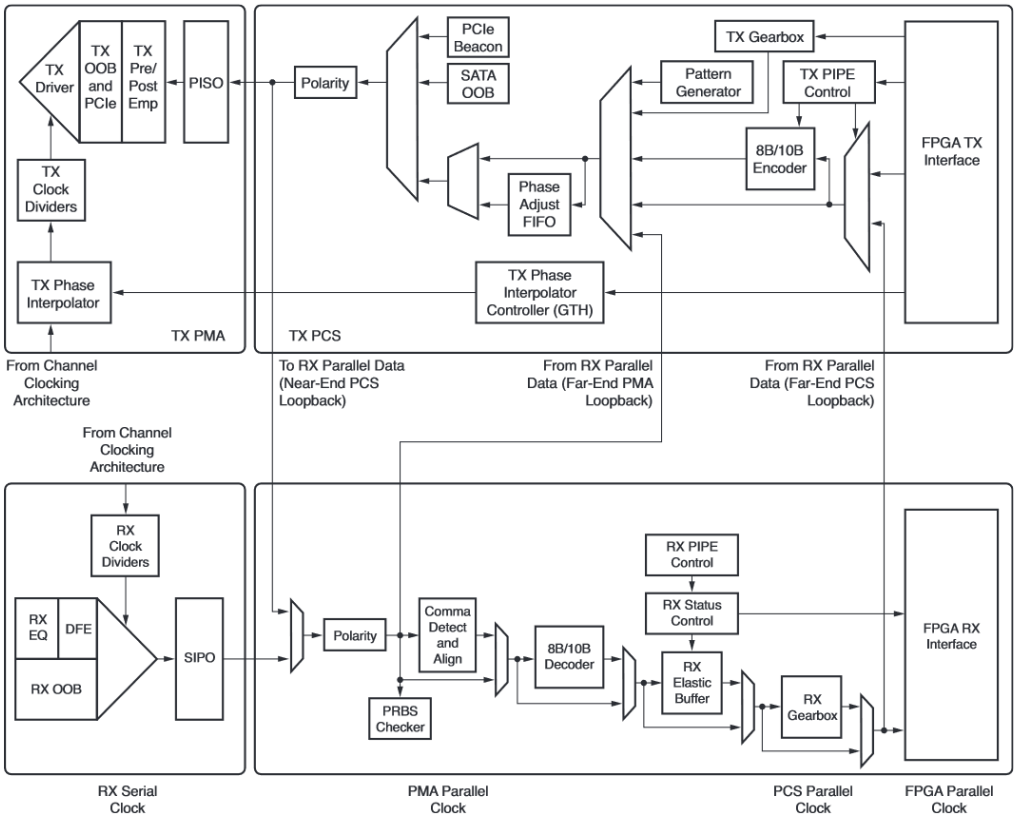

Figure 3-85 Couche PHY des RX et TX d'un transceiver [53]

Cette interface offre une latence minimale d'environ 8 cycles d'horloges. Donc environ 40 ns à 200 MHz pour le transfert d'un paquet. Il reste donc à déterminer comment communiquer du serveur maître aux esclaves.

3.5 Communication du nœud maître vers les esclaves

Dans cette section, nous analyserons la possibilité de faire une carte NIC sur mesure à partir de la VC709, afin d'avoir la plus petite latence possible pour la plus haute bande passante. Le principe

d'évaluation est le même que dans la section précédente. Afin d'avoir notre connexion Ethernet 40 Gbe, nous aurons donc besoin de trois principaux blocs :

- MAC [58] , qui permet de transformer le signal pour qu'il soit dans le protocole Ethernet IEE 802.3.
- PCS/PMA [59] , qui gère tout le traitement du signal afin de pouvoir l'envoyer sur le transducteur puis au connecteur SFP+.
- Transducteurs GTX qui nous permettent l'envoi et la réception des paquets Ethernet sur la connexion.

Afin que la communication soit complément fonctionnelle, les blocs doivent se suivre comme le montre la figure 3-16.

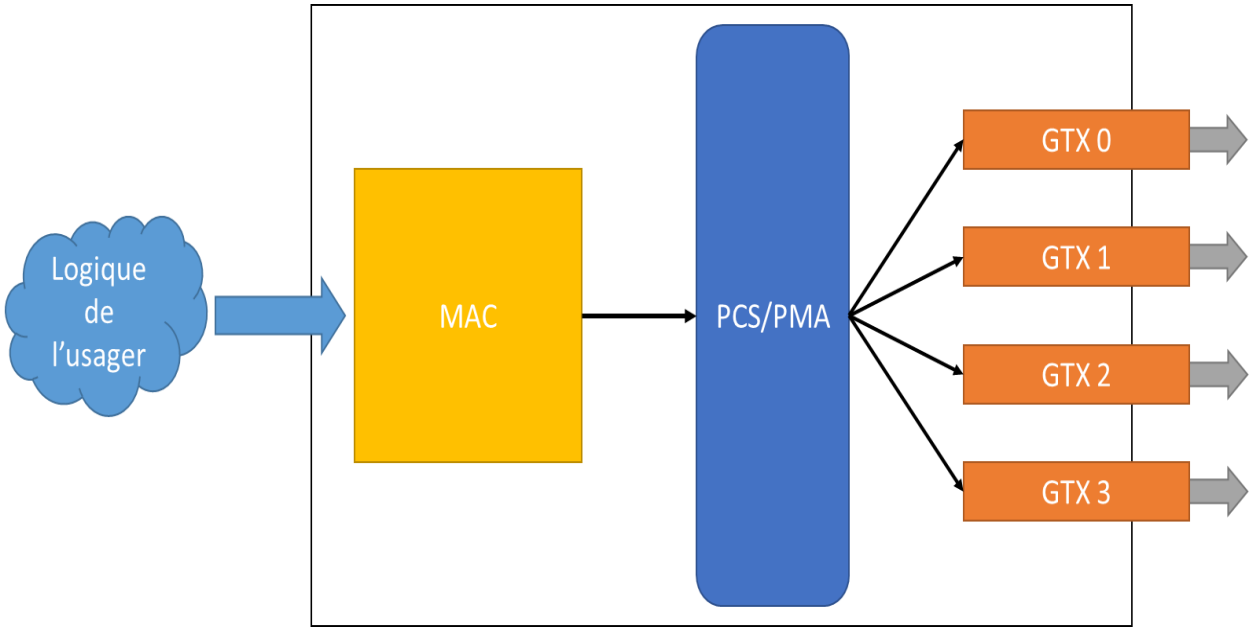

Figure 3-96 Chemin d'un paquet Ethernet 10 Gbe

Alors après examen des différents IP disponibles de la librairie de Xilinx, nous obtenons les valeurs de latences énoncées dans le tableau 3-7, pour une communication Ethernet 40 Gbe.

Tableau 3-7 Latence d'envoi d'un paquet à 40 GBe sur une VC709

	Latence (ns)
MAC IP (pour 64 bits)	38.4
PCS/PMA (pour 64 bits)	62.1
GTX Transducteur	40
Total	140.5

3.6 Étude du système / discussion

Si nous prenons en comptes toutes les valeurs obtenues à travers toutes ces analyses, nous pouvons calculer la latence minimale du système:

$$Latence = L_{CPRI} + L_{CPU_{FPGA}} + L_{FPGA_{FPGA}} + L_{FPGA_{NIC}}$$

$$Latence = \ 0.260 + 1.5 + 0.125 + 0.40 + 0.140 = 2.425\ \mu s$$

Nous obtenons une latence minimale de l'entrée à la sortie de $2.425\mu s$, et nous observons que notre goulot d'étranglement au niveau du transfert de donnée est à la réception des chaines CPRI avec un débit de ligne de 6.144 Gbps.

Nous observons à travers toute cette analyse que la plateforme est viable pour l'implémentation d'une virtualisation de protocoles radio tels que LTE avec une configuration C-RAN. Nous voulions atteindre une latence minimale, en dessous de 5 % du temps total, afin de s'assurer que la latence ne soit pas le goulot d'étranglement de la virtualisation de protocoles. En ce qui a trait la bande passante, nous voulions avoir la plus élevée possible, en nous assurant que nos protocoles de communication ad hoc et propriétaires (QPI) ne soient pas plus lents que les communications conventionnelles, tel que PCIe ou Ethernet. Rappelons que cette section est bornée par une analyse

théorique afin de mettre en lumière les développements impliqués dans la virtualisation de protocoles de communication. Une implémentation de la sorte permettrait d'avoir une plus grande flexibilité au niveau de l'adoption de nouveaux services. C'est-à-dire, que l'utilisation de plus composants accessibles (COTS) réduisent les coûts associés aux mises à jour, sont souvent moins cher que du matériel dédié, et surtout facilite la tâches des concepteurs.

Ajoutons que l'analyse de la grappe se situe au niveau des communications pouvant être les goulots d'étranglement du système. En effet, le transfert des nœuds maîtres vers les esclaves ne fût pas étudié en détail, car nous utiliserons de la fibre optique pour la communication. Ainsi, comme vu à travers le chapitre, les limitations en latence et bande passante proviennent du traitement des données pour l'envoi. Ces limitations dépendent du protocole de transmission par fibre optique choisi. Nous pouvons donc en déduire que cette étude reste pertinente dans le contexte d'une station de base comme celui d'une grappe.

Cette étude de grappe, bien qu'utile comme une preuve de faisabilité théorique, nous servira de base sur laquelle nous bâtirons les chapitres subséquents.

CHAPITRE 4 CONCEPTION D'UN ALGORITHME D'ORDONNANCEMENT

Maintenant que nous savons qu'une architecture de plateforme est capable d'effectuer la virtualisation d'un protocole LTE, nous devons nous concentrer sur l'ordonnanceur et son algorithme d'ordonnancement. Afin d'avoir une bonne idée de la fonction globale d'un ordonnanceur, le chapitre suivant décortiquera l'implémentation d'un algorithme d'ordonnancement au sein d'un système d'exécution, après une brève explication des possibles profils de tâches à gérer. Pour finir, nous explorerons les principales fonctions à implémenter dans un algorithme, puis nous finirons par la conception du plus complexe de ce mémoire, le HEFT.

4.1 Tâches

Dans le domaine de l'informatique, une tâche est aussi appelée une « **unité d'exécution ».** Elle représente une activité devant être accomplie sous des contraintes prédéfinies, tel qu'un laps de temps précis ou même sous un délai lié à d'autres fonctionnalités d'un système. Une unité d'exécution, contient souvent un descriptif de l'activité à exécuter. Elle peut contenir des informations à propos du temps d'exécution, du délai, des échéances à respecter et même des registres à consommer pour effectuer une fonction donnée. Ainsi, une fois la résolution des travaux décrite par une tâche, cette dernière est dite complétée. À travers la littérature, nous retrouvons plusieurs types de tâches.

4.1.1 Périodiques, apériodiques et sporadiques

Comme le nom l'indique, est une **tâche périodique** s'exécute à des intervalles de temps fixes connus. Elles sont donc contraintes à des instances de temps définies. On nomme l'intervalle de temps fixe après lequel une tâche se répète, p_x, la période de la tâche. Si nous considérons T_x comme une tâche périodique, alors le temps à partir de 0 jusqu'à la prochaine instance de T_x est noté $\emptyset_x$, qui représente la phase de la tâche. Ainsi la deuxième instance de la tâche surviendra à $T_x(2) = \emptyset_x + p_x$, donc la troisième à $T_x(3) = \emptyset_x + 2 * p_x$ ainsi de suite.

Les **tâches apériodiques** s'exécutent sur des contraintes de temps aléatoires non prédéfinis contrairement aux tâches à caractère périodique. Ainsi, dans le cas d'une tâche apériodique la séparation minimum entre deux instances consécutives de la tâche peut être 0. Cela signifie que deux ou plusieurs instances d'une tâche apériodique peuvent survenir au même moment.

Les **tâches sporadiques** sont une combinaison des cas de figure périodique et apériodique, où le temps d'exécution est apériodique, mais le taux d'exécution est de nature périodique, du point de vue des contraintes de temps. Les contraintes de temps sont généralement associées à une échéance. Une tâche sporadique T_x peut être représentée par l'équation suivante :

$$T_x = (e_x, g_x, d_x)$$

Où e_x est le temps d'exécution de l'instance de la tâche, g_x représente la séparation minimum entre deux instances consécutives de la tâche, et d_x étant la date buttoir relative. Alors la séparation minimum (g_x) entre deux instances consécutives de la tâche imposent que lorsqu'une instance de tâche sporadique survient, la prochaine instance ne peut subvenir tant et aussi longtemps que l'unité de temps g_x n'est pas écoulée. Ceci restreint donc la fréquence à laquelle une tâche sporadique peut survenir.

La figure 4-1, donne une idée de l'allure de ces tâches à travers le temps.

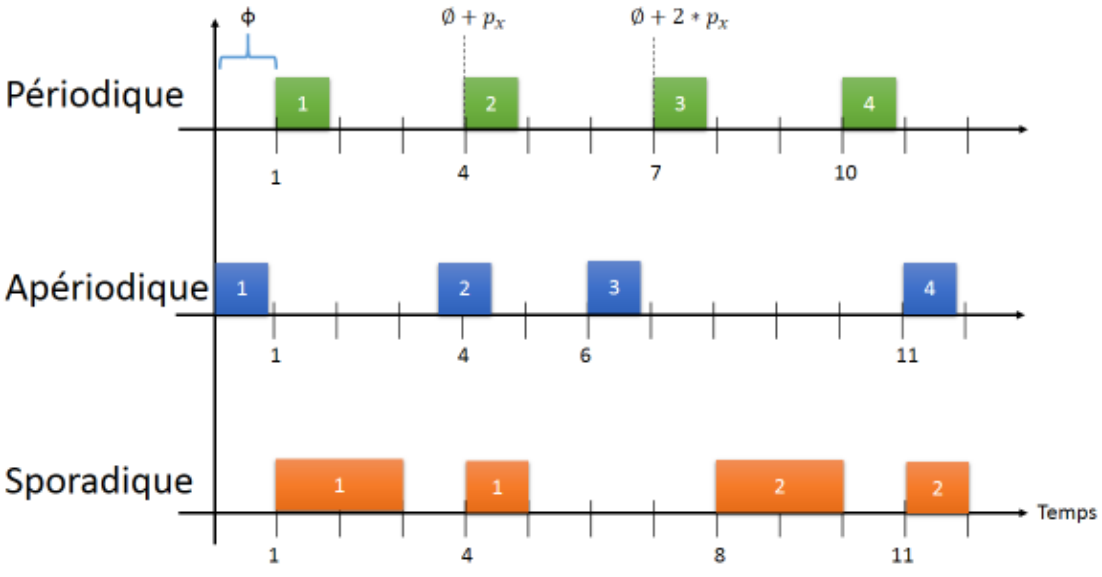

Figure 4-1 Types de tâches contraintes en temps

4.1.2 Avec ou sans préemption

Le phénomène de préemption peut être expliqué à travers un algorithme d'ordonnancement en temps réel où les tâches prioritaires doivent être exécutées en premier. Alors si une tâche est en cours d'exécution et qu'une tâche prioritaire arrive, elle remplacera donc celle en cours, ainsi la tâche en cours est préemptée. Certaines tâches ont la caractéristique de ne pas pouvoir être remplacées, elles sont dites non-préemptives. La préemptivité d'une tâche est spécifique à l'application que le système exécute.

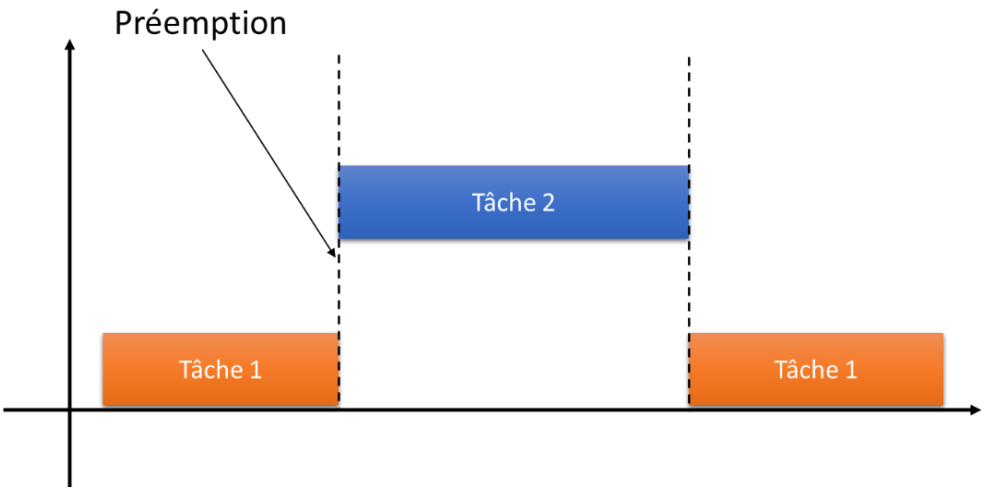

Figure 4-2 Préemption de tâches

4.1.3 Statiques ou Dynamiques

Dans certains systèmes, la priorité de tâche est décidée une fois et ne peut être changée. On parle alors de tâches à caractère statique. Alors que la priorité d'une tâche pouvant être modifiée à tout instant constitue une tâche à caractère dynamique.

4.1.4 Dépendantes et indépendantes

Des tâches peuvent être reliées les unes aux autres. Par exemple, il est possible qu'une tâche ne puisse pas être exécutée tant qu'une autre n'est pas complétée. Elles sont donc dépendantes l'une à l'autre. Il existe deux principaux types de dépendances de tâches, implicite et explicite. Une

dépendance explicite [60] ,que l'on nomme aussi directe, représente un lien de causalité entre les tâches. Par contre une dépendance implicite, aussi appelée indirecte, impose une relation de causalité indirecte entre deux tâches. Une dépendance est causée par le partage d'un espace mémoire commun ou par la communication d'informations d'une tâche à l'autre. Ainsi, afin d'avoir un système performant sous des contraintes temps réelles, un ordonnanceur doit considérer les dépendances de tâches d'une application.

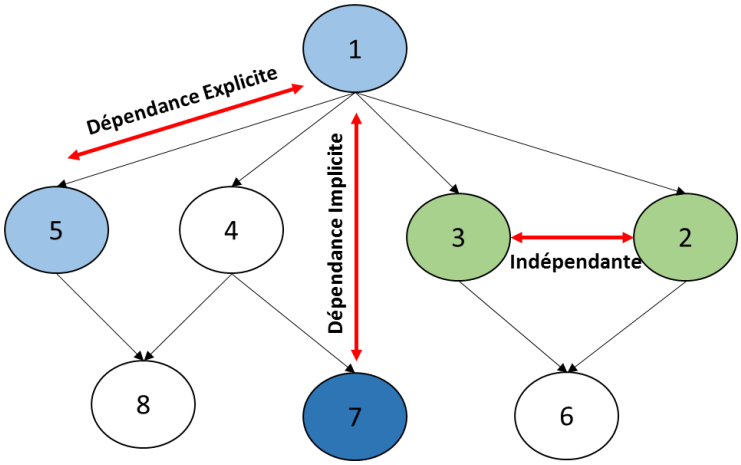

Figure 4-3 Dépendance de tâches

4.2 Caractéristique des algorithmes d'ordonnancement

Nous verrons à travers cette section les différentes caractéristiques des algorithmes basés sur des listes et comment les implémenter dans le système d'exécution StarPU. Définissons tout d'abord où se situe l'algorithme. Il sert à réguler l'accès aux cœurs comme expliqué dans la section 2.2.2.3 de la revue de littérature. La figure 4-4 présente plus en détail le niveau d'abstraction où opère StarPU.

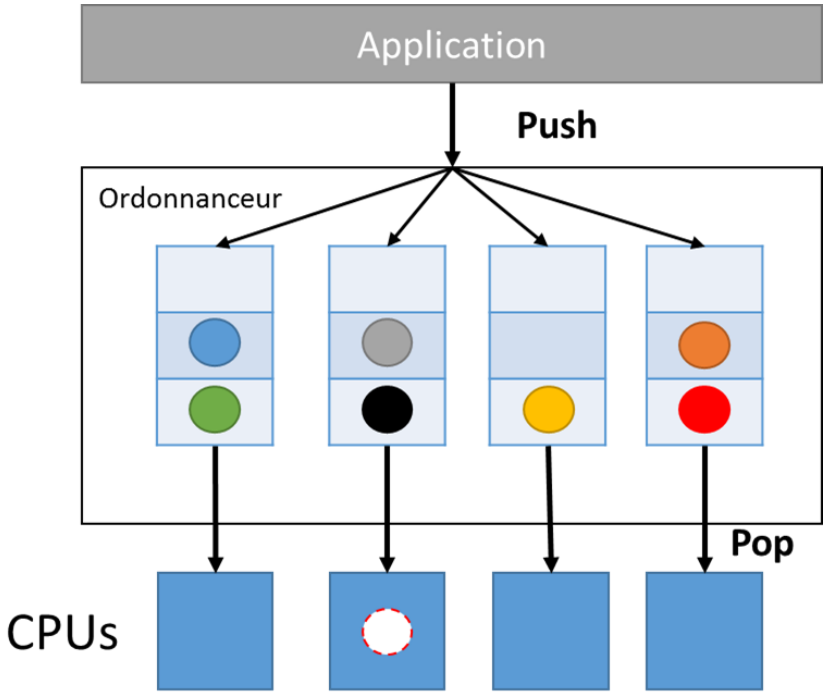

Figure 4-4 Relation entre l'ordonnanceur et les fonctions push et pop

Ainsi, comme nous pouvons le voir dans la figure 4-4, un algorithme possède deux fonctions importantes :

- Le **push**, qui lorsqu'une tâche est marquée comme prête à être exécutée, elle est poussée dans le domaine de l'ordonnanceur pour être associée aux queues des unités d'exécutions en fonction de l'algorithme mis en place.
- Le **pop**, qui se situe à la sortie de l'ordonnanceur, est la fonction qui permet aux unités d'exécution de récupérer les tâches de la queue qui leurs sont associées.

Quand et comment ces fonctions sont utilisées est spécifique à l'algorithme d'ordonnancement. Nous nommons algorithmes à stratégie de push (push based strategy), les algorithmes comme HEFT qui se basent sur quand et comment soumettre les tâches, et ceux basés sur le pop, comme celui basé sur le vol qui pour leur part gère plutôt l'accès des tâches à partir des cœurs. Alors toutes les décisions d'ordonnancement peuvent être prises entre ces deux fonctions. En plus de ces deux fonctions, nous avons les fonctions d'initialisation et de *dé-initialisation* de l'ordonnanceur. Nous

verrons dans les sections subséquentes, comment ces fonctions sont implémentées au sein de StarPU en C.

4.2.1 Fonction d'initialisation

La fonction d'initialisation a pour principal but de mettre en place l'infrastructure pour l'utilisation de l'ordonnanceur. Ainsi la fonction d'initialisation se divise en 5 étapes :

- La création d'une liste d'unités d'exécution qui seront associées à l'ordonnanceur. Une telle liste est nécessaire afin d'avoir un algorithme qui est conscient du contexte dans lequel il se trouve.
- L'allocation de l'espace mémoire nécessaire afin de faire le transfert des données et pour le fonctionnement de l'ordonnanceur lui-même.
- L'initialisation d'une structure de liste pour recevoir et relâcher des tâches, que ce soit à l'entrée de l'ordonnanceur comme pour les listes associées aux unités d'exécution.
- L'association des ressources créées; comme les listes de variables et conditions ; avec le contexte de l'algorithme. C'est donc dire que le programme est au courant de quelles ressources lui sont dédiées (*private*).
- Puis, pour finir, la création du pointeur vers le mutex qui sera dédié à l'ordonnanceur.

Le pseudocode ci-dessous expose comment toutes ces fonctions sont connexes les unes des autres.

```
static void Init_ordonnanceur(algo_id)
{
Création_liste_unités_exec(contexte, liste_unités);
struct Ordonnaceur_data *data= (struct Ordonnaceur_data *)
allocation(taille);
Init_liste_taches(&data->listes_ordonnancement);
Création_contexte(algo_id, (void*)data);
```

```
Init_mutex(&data-> algo_id, NULL);

}
```

4.2.2 Fonction dé-initialisation

La fonction de dé-initialisation est en quelque sorte l'inverse de la fonction d'initialisation. C'est-à-dire qu'elle permet de relâcher les ressources réservées et créées pour l'algorithme d'ordonnancement par son initialisation. Nous avons donc :

- La création d'un espace mémoire qui pointera sur l'ensemble de données crée par l'initialisation de l'ordonnanceur.
- L'effacement de la liste d'unités d'exécutions associées à l'ordonnanceur.
- La destruction du pointeur vers le mutex dédié, comme ça le mutex sera libre d'être alloué à autre chose.
- Puis, la libération des espaces mémoire, comme ça ils pourront être utilisés par d'autres fonctions.

Le pseudocode ci-dessous expose comment toutes ces fonctions sont connexes les unes des autres.

```
static void Deinit_ordonnanceur(algo_id)

{

struct Ordonnaceur_data *data= (struct Ordonnaceur_data *)

get_contexte(algo_id);

Efface_liste_unité_exec(algo_id);

Destruction_mutex(&data->contexte_mutex);

Liberation(data);

}
```

4.2.3 Fonction de push

Dépendant du type d'algorithme utilisé, la fonction de push pourrait être la plus importante. Ainsi c'est au sein de cette fonction que se trouve la majorité de l'algorithme d'ordonnancement. Dans le cas d'un algorithme basé sur la poussée de tâches, les principales étapes sont :

- La réception des tâches dans l'ordonnanceur, par rapport au contexte établi. C'est-à-dire avoir un réceptacle pour les tâches et les faire concorder avec le contexte présent.
- Puis il faut que les fonctions de mutex « lock » et « unlock » soient créées, car lorsque l'on pousse des tâches vers les listes de chaque unité de calcul il faut bloquer les unités de calcul afin de ne pas perde de l'information ou interrompre le transfert.
- Il faut aussi la fonction qui pousse les tâches vers les listes. Une tâche peut être poussée n'importe où au sein d'une liste, il en tient à l'algorithme d'ordonnancement de décider où et quand. Il faut bien sûr s'assurer d'avoir créé un espace dans la liste pour que la tâche puisse être reçue.

Le pseudo-code ci-dessous expose un peu mieux le lien qui existe entre les fonctions discutées plus haut :

```
static int Push_taches(*taches)
{
struct Ordonnaceur_data *data= (struct Ordonnaceur_data *)
get_contexte(algo_id);
mutex_lock(&data->contexte_mutex);
   if (Condition_d'ordonnancement)
   {
    Push_taches_Start_List(&data-> liste_unités, taches);
   }
   Else
```

```
    {
     Push_taches_end (taches);
    }
  mutex_unlock(&data->contexte_mutex);
return 0;
}
```

4.2.4 Fonction de pop

À l'inverse des algorithmes basés sur le « push » de tâches vers les unités de calcul, pour ceux basés sur le « pop » cette fonction devient la plus importante. Les étapes sont donc similaires à celles vues dans la section push au-dessus. Prenons le cas où la fonction pop est secondaire à celle de push. Ainsi nous aurons une fonction simplifiée qui ne servira qu'à récupérer les tâches des listes vers les unités d'exécution. Elle doit donc contenir :

- Encore une fois, une capacité de créer un genre de tampon afin de pouvoir avoir un réceptacle pour la tâche.
- Nous devons aussi implémenter les fonctions de verrouillage et déverrouillage de mutex.
- Pour finir, il faut avoir la fonction qui fait le transfert des tâches vers les unités de calcul. Elle peut être utilisée comme fonction autonome ou être associée à des conditions ultérieures.

Nous verrons grâce au pseudo-code ci-dessous comment utiliser la fonction de pop comme une fonction secondaire à celle de push.

```
struct Ordonnaceur_data *Pop_tache(algo_id)
{
struct Ordonnaceur_data *data= (struct Ordonnaceur_data *)
get_contexte(algo_id);
mutex_lock(&data->contexte_mutex);
```

```
struct Ordonnaceur_data *tache = Pop_taches_Liste(&data->
liste_unités, taches);
mutex_unlock(&data->contexte_mutex);
return taches;
}
```

4.3 Implémentation du Heterogeneous Earliest Finish Time

Après avoir montré en détail les différentes fonctions nécessaires pour implémenter un algorithme d'ordonnancement, nous verrons dans la section suivante l'implémentation complète d'un algorithme complexe utilisé pour les tests. L'algorithme en question est « *Heterogeneous Earliest Finish Time* » (HEFT), qui est basé sur la fonction de poussée de tâches, donc une approche push. Cette implémentation de fonction de poussée est particulière, car elle doit prendre en compte les durées d'exécution espérées pour chaque tâche devant être exécutée. Ainsi en plus des étapes de base qu'une fonction de poussée doit avoir, HEFT contient :

- La création de trois principales listes de tâches supplémentaires :
    - **Temps de départ,** qui correspond à quel moment le début de la tâche semble être planifié pour son exécution.
    - **Temps d'exécution,** qui est le WCET espéré de la tâche. Nous utilisons le pire temps d'exécution pour éviter les cas de sur-ordonnancement qui pourrait causer du chevauchement de tâches, donc ralentir la performance globale de l'exécution de l'ensemble de tâches.
    - **Temps de terminaison,** qui correspond à quel moment la fin de la tâche semble être planifié pour son exécution.
- Le pré-chargement des données d'entrée des tâches devant être exécuté, afin de minimiser le transfert de données entre les unités de calcul.

- Un algorithme déterminant où chaque tâche devrait être acheminé. Ce dernier prend en compte la durée de la tâche, le nombre de tâches dans chaque liste associée aux unités d'exécution et la capacité de chaque unité d'effectuer une tâche efficacement.

Notons, qu'afin qu'un tel type d'algorithme soit efficace, il faut avoir une description la plus complète possible de l'ensemble de tâches, que ce soit par le biais de descripteur détaillé, ou par l'utilisation de modèle de performances. Ces modèles traitent de multiples fois un ensemble de tâches dans le contexte voulant être exécuté et gardent en mémoire la moyenne des temps associés aux tâches. Ce sont des modèles de performance par historique. Il existe d'autres types de modèles de performance pouvant être tout aussi efficace, le principal critère influençant leur rendement est le contexte dans lequel ils œuvrent.

Le pseudocode suivant montre une implémentation typique de HEFT au sein d'un système d'exécution comme StarPU.

```
static double Temps_depart[Max_unites_exec];
static double Temps_exec[Max_unites_exec];
static double Temps_fin[Max_unites_exec];


int HEFT_push(struct starpu_task *task)
{
 for (0 à Nbre_Unités_exec)
 {
  Get_Unité_id(id);
  double meilleur_id, meilleur_fin, meilleur_exec;
  double longueur = get_temp_exec(tache, id);
  double Temps_fin = Temps_fin[i] + longueur;
```

```
  if (Meilleure Exécution)
  {
    meilleur_id = id;
    meilleur_fin = Temps_fin;
    meilleur_exec = longueur;
  }
}
mutex_lock(&contexte_mutex[meilleur_id]);
Temps_exec[meilleur_id] += meilleur_exec;
Temps_fin[meilleur_id] += meilleur_exec;
mutex_unlock(&contexte_mutex[meilleur_id]);

unsigned noeud_memoire = get_noeud_memoire(meilleur_id);
préchargement_données_entrées(tache, noeud_memoire);

return Push_taches_local(meilleur_id, tache);
}
```

CHAPITRE 5 EXPÉRIMENTATION

L'ensemble du contexte ayant bien été défini à travers la revue de littérature, le chapitre suivant se situe à un point spécifique de la virtualisation d'une pile LTE. En effet, nous nous concentrerons sur la gestion des tâches associées à la virtualisation. Les questions posées sont :

- Est-ce que l'ordonnancement de tâches est faisable sur un simple cœur d'i7 sous les contraintes de temps imposées par le protocole?
- Est-ce que l'algorithme d'ordonnancement offre une bonne durée d'exécution et utilisation du processeur?
- Et enfin, est-ce qu'un processeur doit être dédié à la gestion de tâches?

Les sous-parties ci-dessous exposeront les résultats obtenus et les méthodes utilisées à travers nos expérimentations et tests.

5.1 Spécifications des tests

La section 2.3.3 donne une vue générale de la pile LTE, cependant pour que les tests soit complets, nous avons besoin d'un niveau de détail accru permettant d'évaluer les besoins associés à chacune des trois étapes de la virtualisation d'une pile LTE. Une fois fait, nous aurons une description complète du problème à évaluer.

Notons qu'à travers ce chapitre nous travaillons sous les contraintes suivantes :

- Nous prenons le pire cas, car si le système est fonctionnel il devrait l'être dans tous les cas.
    - Un usager utilisant toutes les ressources accessibles (Single High Data Rate User) [42] :
        - Utilise 100 blocs de ressources : 1200 porteuses
        - Bande passante du protocole : 20 MHz
        - Débit de données : 300 Mbps
        - Cas à 4 antennes (MIMO x4)

- Le traitement des données analogiques et RF est pris en compte en amont.
- Ne traite que la couche PHY de l'Uplink

Notons qu'à travers les analyses suivantes se basent sur les études faites par Frigon et al[42].

5.1.1 Traitement frontal

Le traitement frontal (Front-End Processing) a pour principale tâche de convertir les données reçues des antennes du domaine temporel vers le domaine fréquentiel, afin que le bloc suivant puisse utiliser les informations transmises. La bande passante du protocole de 20 MHz impose que le traitement OFDM équivaille à faire un des FFT de 2048 points. La FFT étant la fonction la plus exigeante en puissance de calcul des processeurs, nous la prendrons comme base pour notre évaluation. Prenons un intervalle de temps de 500 ns, durant ce dernier un BBU virtuel peut recevoir jusqu'à 7 blocs de 2048 points. Ainsi, sur un système comportant des cœurs i7 d'Intel associés à une bibliothèque de calcul, comme Math Kernel Library (MKL), un cœur peut traiter une FFT représentant 71.4 µs d'échantillons reçus en seulement 15 µs sur un seul processeur. Donc, si nous gardons les contraintes énoncées au-dessus, et que nous utilisons une fréquence d'échantillonnage de 30.72 MHz avec 16 bits de précision, le pire cas de 1200 porteuses produira 81920 bits par antennes, et sortira 153 600 bits par bloc OFDM de 71.4 µs. Cela signifie que le traitement frontal aura comme coût les temps cités au Tableau 5-1 si nous utilisons 1 cœur d'i7 :

Tableau 5-1 Caractéristiques du traitement frontal MIMO

	Taille de données consommées	Taille de données en sortie	Temps de réception	Temps de traitement
1 antenne	81 920 bits	153 600 bits	71.4 µs	15 µs
2 antennes	163 840 bits	307 200 bits	142.8 µs	30 µs
4 antennes	327 640 bits	614 400 bits	285.6 µs	60 µs

Cependant, ces temps peuvent être améliorés d'un facteur égal au nombre d'antennes, si nous utilisons plus de cœurs pour le traitement frontal, ainsi nous pourrions traiter les 4 antennes en même temps, créant donc une latence de 71.4 µs pour la réception à laquelle s'ajoute un temps de traitement des FFT de 15 µs. Comme l'expose la figure 5-1.

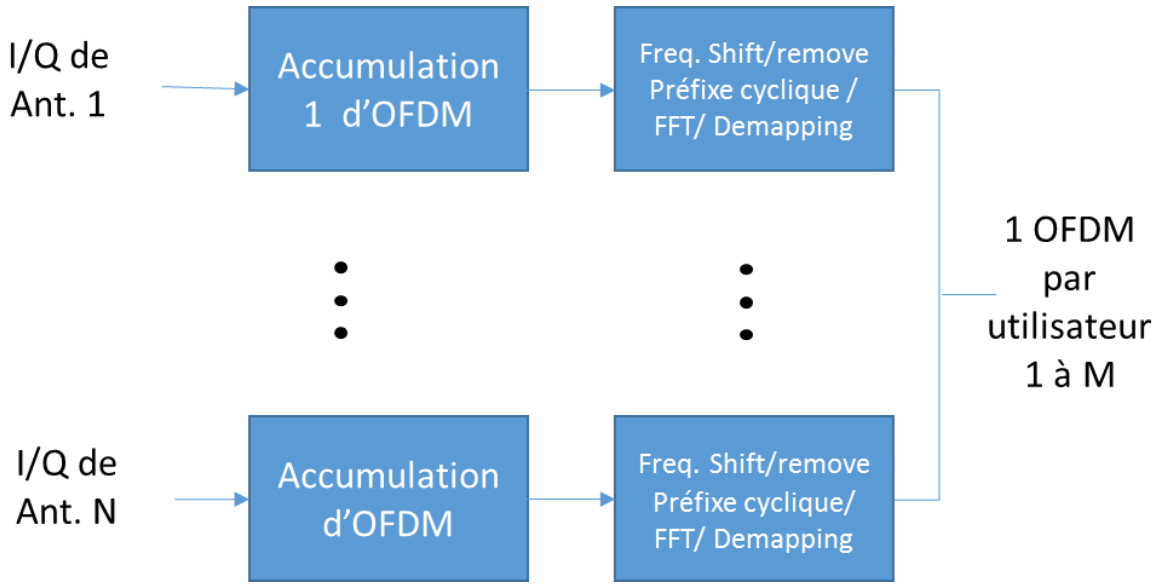

Figure 5-1 Représentation du traitement parallèle des OFDMs

5.1.2 Traitement du signal

Le traitement du signal par utilisateur (*Per-user signal processing*) vise à transformer le signal reçu par le traitement frontal pour le convertir en un format de données adéquat afin que le bloc suivant puisse l'utiliser. Les principales tâches devant être effectuées sont :

- Faire l'estimation de canal pour chaque antenne et le calcul de la matrice d'égalisation multi antennes de la sous-porteuse (subcarrier).
- Faire l'égalisation des multi-antennes par porteuse (carrier).
- Calculer l'IFFT pour toutes les sous-porteuses associées à un utilisateur.
- Corriger le décalage en fréquence et phase.
- Effectuer l'analyse de la constellation (ex : 64-QAM) et le débrouillage de symbole (*Descrambling*)

Nous observons que l'opération qui demande la plus de puissance de calcul est l'IFFT. Ainsi, restons dans les mêmes contraintes stipulées au-dessus. Pour le « pire cas », ou nous avons un Single High Data Rate User, la bibliothèque MKL permet d'effectuer une IFFT en 10 µs pour le cas d'un intervalle de temps de 0.5 ms, 8 IFFT doivent être effectuées, deux pour l'estimation de canal et une pour chaque symbole d'OFDM. Cela équivaut à 0.320 ms pour le traitement de toutes les IFFT sur un cœur d'i7.

Pour la prochaine étape, l'égalisation des 4 antennes (MIMO) est estimée à 1 µs par sous-porteuse lorsqu'exécuté sur un i7, il faudra donc 1.2ms le cas échéant. Nous observons que nous dépassons le temps disponible dans notre intervalle, il faut donc recourir à la parallélisation. Il faudra donc au minimum 4 cœurs pour rentrer dans l'intervalle de temps alloué, permettant d'avoir une exécution de 0.38 ms au lieu de 1.52 ms.

Donc pour 100 utilisateurs, nous recevons du bloc frontal 153 600 bits à l'entrée et le traitement par utilisateur crée 18 424 bits par utilisateur, équivalent à 1 842 400 bits en tout.

5.1.3 Décodage

Le décodage du signal (*Per-user decoding*) consiste à transformer le signal pour être traitable par la couche supérieure du protocole, la couche MAC. Le Décodeur Turbo est la tâche la plus longue à exécuter dans la chaine de décodage. Le décodage divise les données d'entrées en blocs de transport. L'exécution du pire cas, avec un usager exploitant 1200 sous-porteuses se traite en 10 ms avec une taille de bloc de 299 856 bits sur un coeur de i7.

Ainsi, le traitement de toute la chaine d'Uplink de 4 antennes sur un cœur d'i7 prendrait :

$$Traitement\ Uplink = Réception + 4\ \times Traitement\ OFDM\ + per_{user} + décodage$$
$$= 500 + 71.4 \times 4 + 15 \times 4 + 1520 + 10000 = 12365{,}6\ \mu s$$

Il faut savoir que le protocole LTE impose d'avoir une réponse de reconnaissance (*acknowledge*) 3 ms après la fin du premier intervalle de 1ms du traitement des sous-porteuses d'un usager. Nous devrons donc paralléliser le tout pour rentrer dans notre intervalle de temps. Afin de rencontrer les temps, nous devrons au moins avoir la configuration de cœurs de calcul, énoncés dans le tableau 5-2.

Tableau 5-2 Nombre de cœurs d'i7 requis pour le traitement du Uplink

	Traitement Frontal	Traitement du signal	Décodage
Nombre de cœurs	1	4	10
Temps	345.6 µs	380 µs	1ms

Un tel agencement de cœurs, nous permettra de traiter la chaine d'Uplink en 2225.4 µs laissant donc 1774.6 µs pour le reste des traitements. Cependant, nous supposons que la gestion des tâches sera faite avec l'utilisation d'un cœur d'i7 en plus. Une question intéressante est de savoir si l'ordonnancement de toutes ces tâches est faisable sans dépasser nos contraintes de temps imposées par le protocole. Les parties suivantes expliquent les tests effectués afin de caractériser la gestion des tâches dans la situation décrite au-dessus.

5.2 Méthode de test

Cette section se découpe en deux principales parties. Dans la première, nous décrivons l'environnement de test avec précision, car la compréhension de cet environnement est à la base des résultats obtenus. Nous décrivons notamment la configuration, le matériel ainsi que les bibliothèques choisies pour effectuer les tests. Puis, la seconde partie expose le plan de test qui décrit comment le test est fait dans son ensemble, associé avec des explications spécifiques sur comment les fonctions utilisées sont implémentées.

5.2.1 Environnement de test

Nous commencerons par décrire le matériel sur lequel les tests ont été effectués, puis les outils utilisés pour bonifier notre application.

5.2.1.1 Machine virtuelle et Ordinateur

La preuve de concept a été faite sur un ordinateur portable avec les ressources suivantes :

- Processeur i7-4700HQ 64 bits cadencé à 2.40GHz, qui possède 4 cœurs physiques doubles filaires (double threaded), qui permettent de disposer de 8 cœurs logiques pouvant être utilisés comme des unités de calcul indépendantes. Le tout communiquant avec 6 MB d'antémémoire.
- 24 Go de mémoire vive associée au CPU.

Afin d'avoir le plus de contrôle possible sur le matériel, nous avons utilisé une machine virtuelle, grâce au logiciel Virtual Box, sur laquelle nous avons installé la version de Linux, Ubuntu 14.04.

5.2.1.2 Bibliothèques

Comme nous le verrons plus loin, nous utilisons StarPU et DPDK qui sont deux bibliothèques fournissant des fonctionnalités des services et fonctionnalités complémentaires dont l'usage coordonné contribue à atteindre les objectifs visés de performance et de faible latence dans la gestion d'un ensemble des tâches sur un processeur multi-coeur-multi-filaire.

Donc mis à part les principales fonctions de C, StarPU [61] comme système d'exécution permet :

- De gérer les dépendances Explicites/ Implicites.
- De gérer l'accès aux processeurs sur la puce i7.
- D'avoir une programmation basée sur les tâches sous plusieurs contextes en simultané.

Jumelé à StarPU, nous aurons deux outils compatibles avec la version 4.6 de gcc. FxT qui est un outil permetant d'avoir des traces (le moins invasives possible) afin de pouvoir caractériser le mieux possible l'application exécutée. D'autre part, nous utilisons aussi l'outil HWLOC, qui permet d'avoir de l'information sur le matériel de la machine utilisé en temps réel.

Enfin nous utiliserons le Data Development Kit (DPDK) d'Intel pour :

- Faire des allocations de mémoire efficaces par le biais de *hugepages*.
- Avoir une infrastructure de mémoire tampon.
- Et surtout, pour avoir plus de contrôle sur les processeurs.

La plateforme de test est illustrée par la figure 5-2.

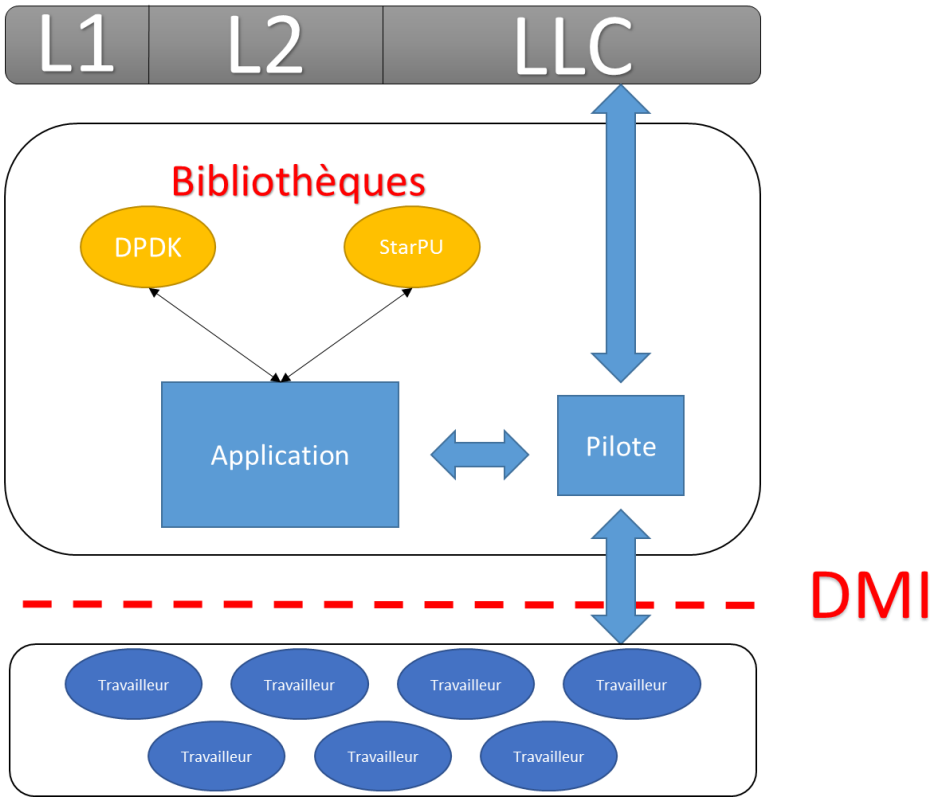

Figure 5-2 Plateforme de tests

5.2.2 Plan de test

Comme expliqué dans les sections précédentes du chapitre, si nous voulons respecter les contraintes de temps associées au protocole LTE, la plateforme doit avoir de multiples cœurs de processeurs disponibles et associés à la bibliothèque MKL d'Intel. N'ayant pas accès à ce genre de plateforme, nous devons simuler l'utilisation de la bibliothèque de calcul sur notre système.

Premièrement nous devons définir le profil des tâches. Nous savons que le protocole LTE contient un grand nombre de tâches ayant une faible durée, entre 1 et 10 µs. Ajoutons aussi que dans le traitement de l'Uplink, nous avons un parallélisme élevé mais une dépendance entre les blocs. En effet, lorsque nous traitons les tâches d'un même bloc, nous pouvons les exécuter en parallèle, mais le prochain bloc ne peut commencer tant que toutes les tâches du précédent ne sont pas terminées.

La dépendance de tâches sera gérée par le biais du système d'exécution StarPU, ce qui nous permet d'utiliser des entités appelées « tag ». Le concept est exposé à la figure 5-3.

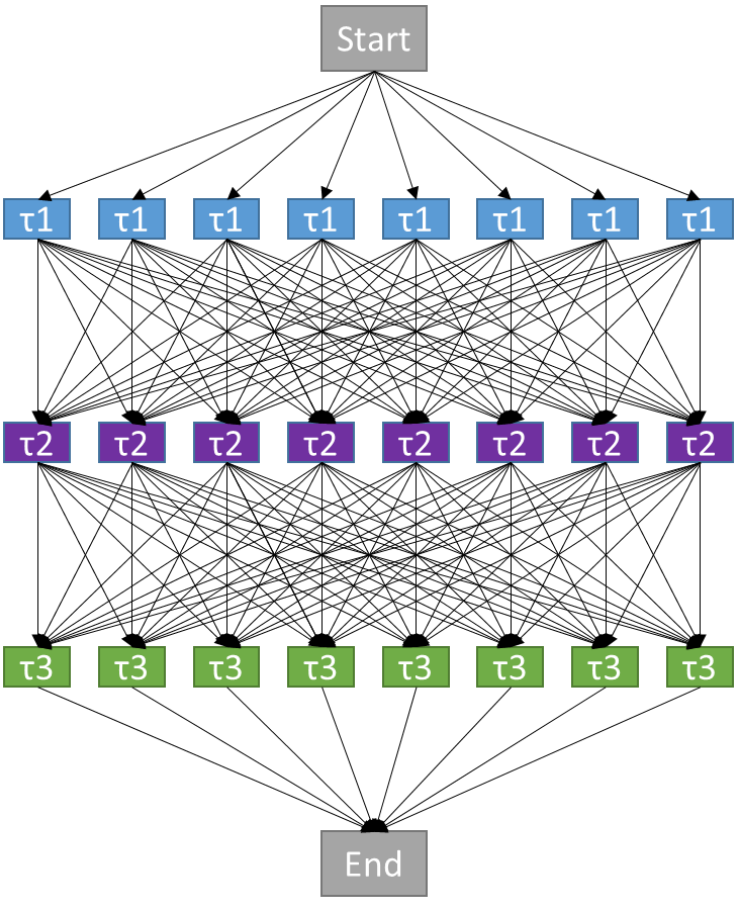

Figure 5-3 Représentation des dépendances de tâches

Deuxièmement, nous devons avoir une technique d'émulation des fonctions accélérées par MKL. Dans les tests, nous allons traiter ces fonctions comme des boites noires. Nous avons estimé plus haut la durée d'exécution de chacune des fonctions. Nous aurons donc un ensemble de tâches qui représentent chaque bloc et qui une fois combinées auront une durée équivalente à celle d'une fonction exécutée avec la bibliothèque de calcul. Ainsi, le nombre de tâches associées à chaque fonction sera dépendant de deux principaux facteurs, la durée d'exécution de chaque tâche et le nombre de cœurs de calcul de la machine.

La durée d'exécution sera gouvernée par la durée de la fonction associée à chaque tâche et le nombre de tâches. Comme nous le voyons, les deux ont une relation symbiotique, c'est-à-dire que

l'un affecte l'autre. La durée de la fonction sera faite par le biais d'un « usleep(x) », et la valeur de x sera faite aléatoirement dans un intervalle de 1 à 10 µs. Prenons l'exemple du bloc de traitement du signal, si nous imposons que chaque fonction associée à une tâche dure 1 µs, nous devrions avoir 380 tâches si nous les exécutions sur un cœur. Alors la durée cumulée de ces tâches serait équivalente à l'exécution du bloc avec l'aide de MKL. Cela nous mène à la deuxième étape de l'exécution des tâches, à savoir la majoration du nombre de tâches en fonction du nombre de cœurs de calcul. Cette majoration est primordiale si nous ne voulons pas modifier le comportement du protocole sous les contraintes mises en place. Ainsi, si nous reprenons le cas du dessus pour lequel nous supposons qu'il y a 380 tâches avec un cœur d'exécution, nous devrions avoir 760 tâches avec le même profil si nous avions deux cœurs de calcul. En effet, nous devons prendre le meilleur scénario de parallélisation, si les deux cœurs exécutent les 760 tâches en même temps, à la fin de la chaine, nous aurions eu la même durée que si nous avions 1 cœur pour 380 tâches. Grâce à cette méthode, nous nous assurons que des délais supplémentaires ne sont engendrés que par les accès mémoire et surtout, l'algorithme d'ordonnancement. Le tableau 5-3 expose l'évolution du nombre de tâches en fonction du nombre de cœurs disponibles sur une machine, toujours avec la même configuration de tâches.

Tableau 5-3 Nombre de tâches en fonction du nombre de cœurs

	Traitement Frontal	Traitement du signal	Décodage
1 cœur	346	380	1000
2 cœurs	692	760	2000
4 cœurs	1384	1520	4000
6 cœurs	2076	2280	6000

Ensuite, nous devons définir le plan de test. Nous savons que le protocole induit une dépendance de tâches comme le montre la figure 5-3. Le test sera donc fait comme une chaine d'Uplink, c'est-

à-dire nous que nous commençons par une émulation des délais de réception sur les antennes. Puis, la chaine de traitement est commencée. Nous aurons alors 3 blocs tous dépendants du précédent, et chacune des tâches intrinsèques aux blocs sera indépendante, donc on aura des exécutions très parallèles. Le schéma suivant expose le plan de test en son ensemble. Chaque bloc aura son profil de tâches dédié. En effet, le système d'exécution nous permet d'établir des contextes, ils définissent les fonctions que les tâches pourront accéder et la plateforme sur laquelle elle devrait s'exécuter. Cette entité de configuration s'appelle un codelet.

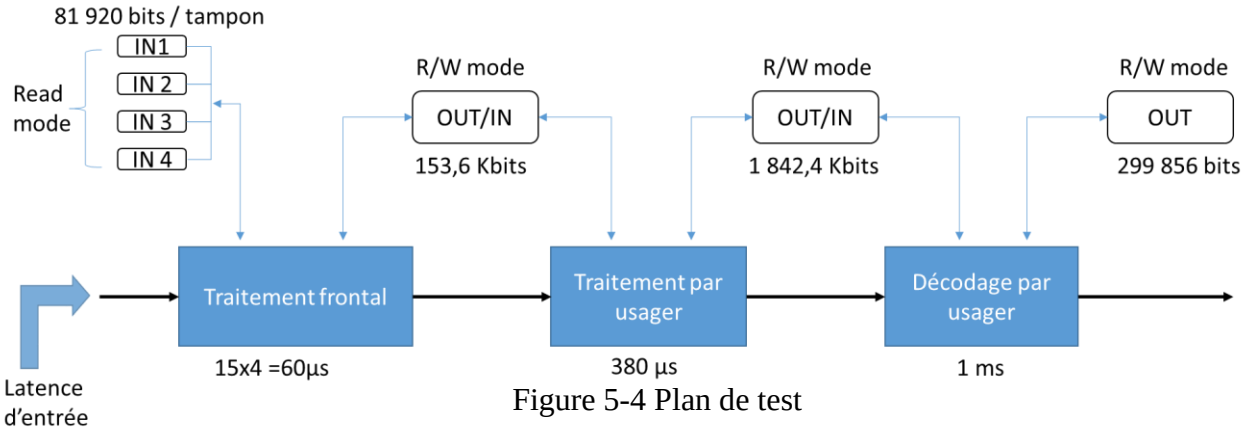

Figure 5-4 Plan de test

Afin d'avoir un test plus significatif, nous devons ajouter un aspect important de la gestion de tâches, l'accès aux données. En effet, le temps associé à l'acquisition et à l'écriture de données est l'un des goulots d'étranglement lors de l'ordonnancement de tâches, car il doit gérer les dépendances de données ainsi que le partage de ces dernières entre chaque unité de calcul. Tout d'abord, nous utiliserons la bilbiothèque DPDK pour l'allocation de mémoire. Elle nous permet, par le biais de ses API, telles que rte_malloc et rte_mempool, d'avoir de la mémoire tampon localisable dans des hugepages de taille assez grandes pour supporter la quantité de données devant être lues et écrites. Une fois la mémoire allouée par DPDK, nous utiliserons StarPU pour créer des tampons de taille spécifiques. Le comportement de ces tampons sera défini au sein du même codelet qui configure le profil de tâches de chaque bloc. Le codelet nous permettra de définir le mode d'accès de la mémoire tampon que nous voulons utiliser. Trois principaux modes sont disponibles, STARPU_R qui permet une mémoire en lecture seule, STARPU_W qui permet une mémoire en écriture seule et STARPU_RW qui permet un tampon pouvant être lu et écrit. Le code ci-dessous

expose un codelet comportant 3 blocs de mémoire tampon définis pour exploiter les différents modes.

```
struct starpu_codelet cl = {
.cpu_funcs = {cpu_dummy_func, NULL},
.cuda_funcs = {gpu_dummy_func, NULL},
.nbuffers = 3,
.modes = {STARPU_R, STARPU_W, STARPU_RW},
};
```

Nous devons ajouter qu'afin d'avoir un test plus déterministe, nous associerons les tâches aux tampons non seulement par le biais de son utilisation par une fonction, mais aussi avec l'aide de « *data handle* ». Il nous permet de spécifier pour un bloc mémoire quelles tâches a le droit d'y accéder. Le test aura donc 7 tampons, 4 d'entrées de 81 920 bits de taille en mode lecture, représentant le flux de donnés des OFDM partant des antennes vers le système. Puis trois en mode lecture et écriture :

- Un de 153 600 bits, représentant les données de sortie du bloc frontal, mais l'entrée du bloc de traitement du signal,
- Un autre de 1 842 400 bits, qui est la sortie du bloc de traitement du signal et l'entrée du bloc de décodage,
- Puis un de 299 856, qui sont les données de sortie du décodage que l'application devra acheminer vers la couche supérieure, MAC.

Le plan de test dans son ensemble se passe en deux parties. Le premier a trait directement à l'ordonnancement des tâches. Nous évaluons tout d'abord si l'ordonnancement sur cette plateforme est assez efficace pour rencontrer les contraintes du protocole. Ensuite, nous testerons plusieurs algorithmes d'ordonnancement différents afin d'évaluer lequel serait le plus adéquat pour notre application. Les algorithmes, définis avec plus de détail dans la section 2.2.3, seront :

- Glouton,

- Chaines de priorités (Prio)
- De vol (Work stealing)
- HEFT (heterogeneous earliest finish time)
- Dmda (deque model data aware)
- Aléatoire

Puis nous finirons l'ensemble de tests avec une vérification de l'impact de l'algorithme sur la gestion de tâches d'une chaine d'un lien montant.

La prochaine section exposera les résultats obtenus à travers les tests énoncés au-dessus.

5.3 Résultats

Comme discuté dans la section précédente, nous diviserons l'exposition des résultats en deux parties, l'ordonnancement et l'impact de l'ordonnanceur sur les cœurs de calculs.

5.3.1 Ordonnancement

Pour tenter de résoudre la première hypothèse exposée au début de ce chapitre, nous avons fait fonctionner l'application définie par la section méthode de test 5.2, et ce pour chacun des algorithmes énoncés au-dessus. L'histogramme de la figure 5-5 expose les résultats obtenus, les valeurs sont en millisecondes et ne comportent pas le délai de réception des antennes de 500 µs.

Nous observons que l'algorithme glouton est le plus efficace en terme de durée d'exécution prenant au total 2.71 ms, laissant donc une marge assez grand pour effectuer le reste du traitement de la chaine montante et celle de la chaîne descendante.

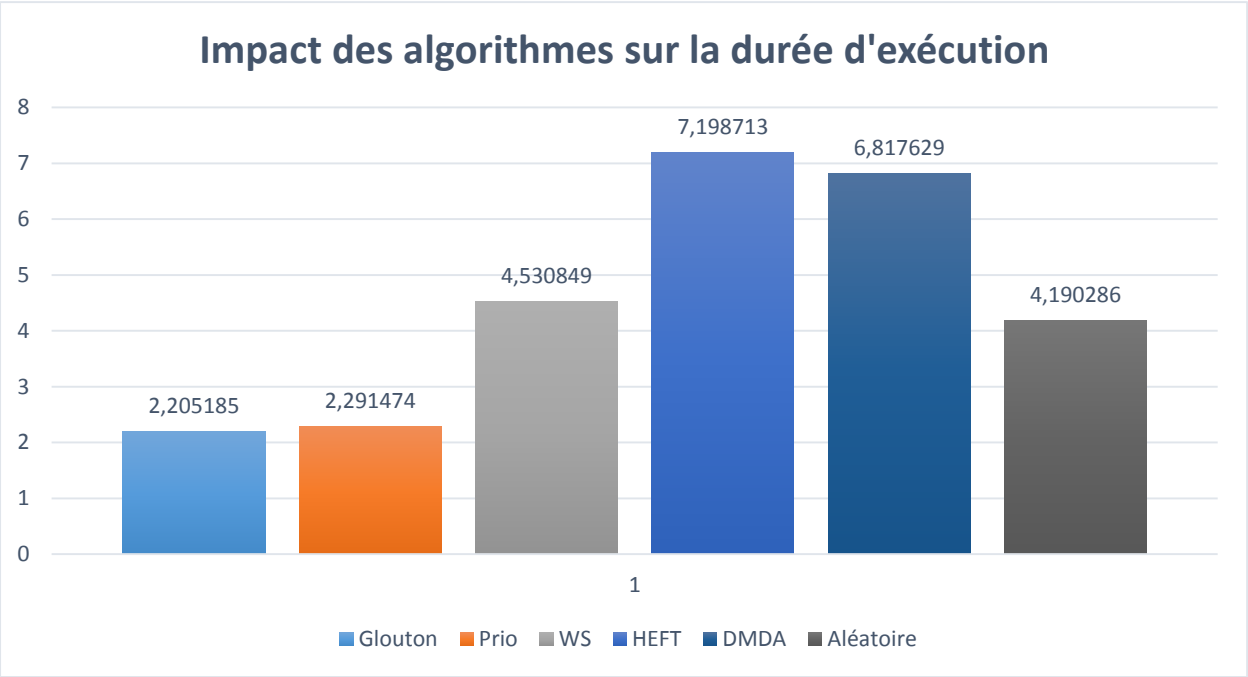

Figure 5-5 Impact des algorithmes sur la durée d'exécution

Cependant, comme le montre le tableau 5-4 plusieurs algorithmes dépassent la marge allouée des 4 ms. Nous verrons plus en détail l'impact de la complexité de l'algorithme dans la section suivante.

Tableau 5-4 Durée totale d'exécution des algorithmes

	Glouton	Priorités	De Vol	HEFT	DMDA	Aléatoire
Durée totale (ms)	2,71	2,79	5,03	7,70	7,32	4,69

5.3.2 Impact des ordonnanceurs

Dans la section suivante, nous allons décortiquer les différentes techniques d'ordonnancement, ainsi que leurs impacts sur la gestion des tâches. Afin de pouvoir étudier la figure 5-6, représentant les allures de l'ordonnanceur sous la forme d'histogrammes, nous devons adéquatement définir sa légende. Nous avons sept états dans lesquels les cœurs peuvent se trouver :

- **Initializing**, correspond à la mise ne place de l'infrastructure de test, c'est-à-dire les allocations d'espaces mémoire et tout ce qui a trait à l'algorithme d'ordonnancement.
- **FetchingIn**, correspond à l'action de récupération de données nécessaire pour l'exécution des tâches par l'unité de traitement (Worker). Ainsi le travailleur va chercher les données d'entrée.
- **PushingOut**, survient lorsqu'une unité de calcul a terminé l'exécution d'une tâche. Il se débarrasse donc de la tâche pour pouvoir en traiter un autre.
- **Ordonnancement**, correspond à toutes les actions liées à l'algorithme d'ordonnancement, de la latence de l'algorithme lui-même jusqu'aux actions « push » et « pop » vers les unités de calculs.
- **Overhead**, équivaut au temps passé à créer une tâche, l'attribuer aux cœurs, et aussi le temps passé à l'arrêt ou la pause des cœurs lorsque la tâche est terminée. C'est-à-dire une fois que la tâche soit passée par l'étape du « PushingOut ». L'overhead de chacun des ordonnancements devrait être similaire vu que nous utilisons les mêmes profils de tâches ainsi que les mêmes unités de calculs.
- **Dormant**, est le terme utilisé pour dire que le cœur est en veille, c'est un moment où l'unité de calcul ne fait rien et est en attente pour une prochaine action.
- **Exécution**, est le temps de traitement des tâches et des actions associées à cette dernière. Sur les histogrammes, les termes « dummyA/B/C » correspondent à chaque groupe de tâche de la chaine de l'uplink.

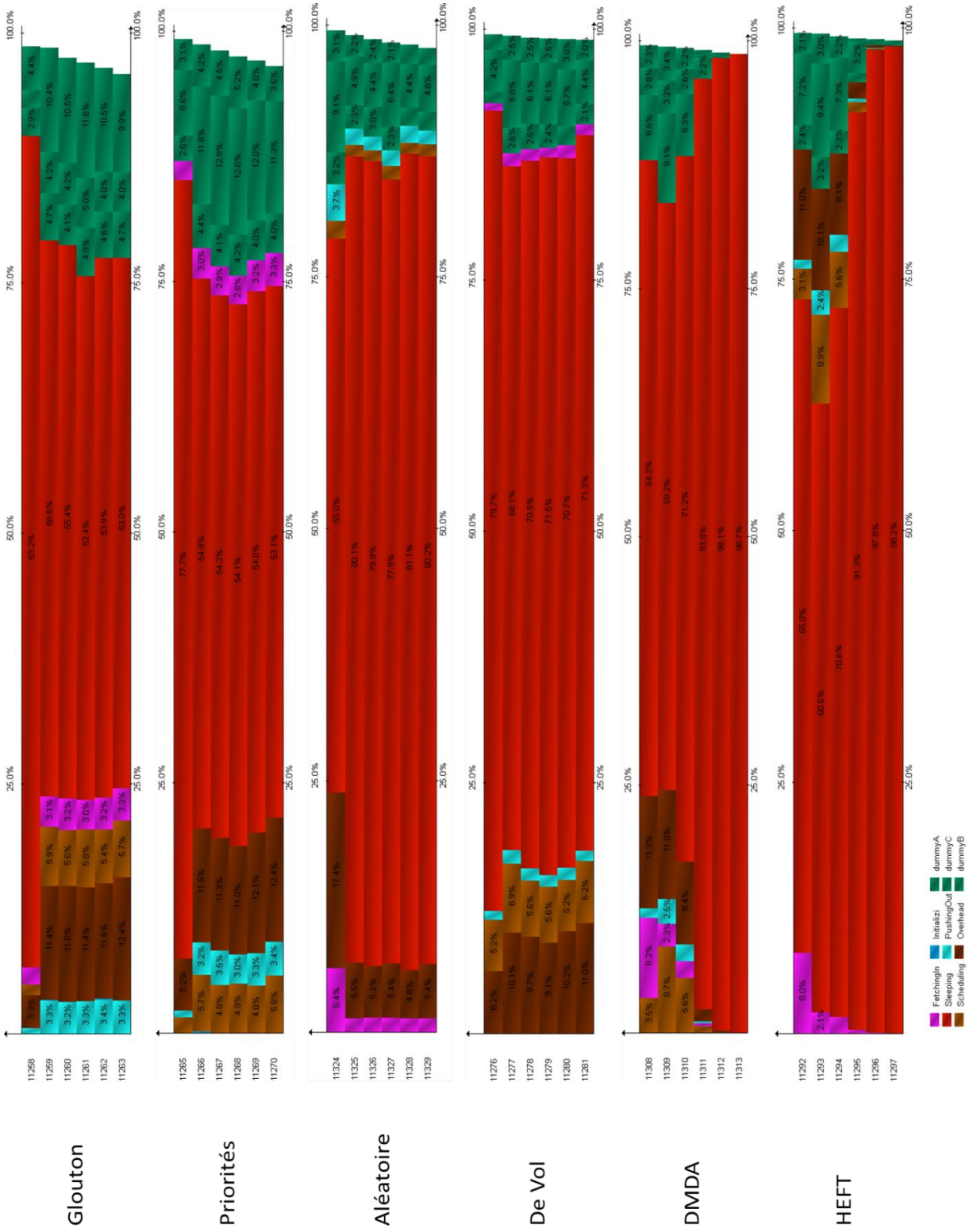

Figure 5-6 Histogrammes d'exécutions sur les unités de calculs

Nous observons sur la figure 5-6 que chacune des étapes semble similaire à travers les différents algorithmes d'ordonnancements. Ainsi, afin d'avoir une meilleure vue d'ensemble nous allons soutirer les informations de la figure 5-6 et les transposer dans le tableau 5-5. Notons que les rapports d'inefficacité se basent sur la durée d'exécution avec l'algorithme le plus rapide, dans notre cas il s'agit du glouton. Ainsi pour avoir une comparaison adéquate entre chaque algorithme, nous devons analyser leurs rapports d'inefficacité.

Tableau 5-5 Récapitulatif des effets des ordonnanceurs

	Overhead (%)	Ordonnancement (%)	Exécution (%)	Dormant (%)	Rapport d'inefficacité
Glouton	10.28	4.916	17.73	60.58	1
Priorité	10.583	4.6	19.183	58	1.039
Aléatoire	7.283	1.316	10.3	75.7	1.900
De vol	9.383	5.783	10.03	71.96	2.055
DMDA	5.366	3.1	7.033	80.86	3.128
HEFT	5.216	3.13	7.483	80.58	3.264

Rappelons que l'objectif énoncé lors de l'introduction de ce chapitre est de trouver un algorithme d'ordonnancement qui offre un bon compromis entre la durée d'exécution et l'utilisation des unités de calcul sous des contraintes LTE. Tout d'abord nous observons que les « overheads » associés à chacun des algorithmes sont similaires, pour le voir il suffit de voir la corrélation entre le rapport d'inefficacité et le pourcentage d'utilisation des cœurs par l'overhead. Ce comportement est adéquat, car il s'agit du même profil de tâches exécuté sur la même plateforme.

Afin de pouvoir résoudre cette question de recherche, nous devons nous fier aux trois facteurs restants du tableau. Commençons par l'ordonnancement. Nous observons que l'algorithme le plus simple et le moins demandant en temps d'ordonnancement est l'aléatoire. En effet, ce dernier

consiste juste à distribuer les tâches lorsque reçues vers n'importe quelle queue d'unité d'exécution. Cet algorithme est donc de basse complexité avec une utilisation de cœurs d'environ 42 %. À prime abord cet algorithme pourrait être considéré comme adéquat, cependant comme son nom le suggère, il effectue un ordonnancement aléatoire. Vu que les statistiques correspondantes à l'algorithme furent obtenues en faisant la moyenne de milliers de tests, nous avons pu constater que certaines fois nous avions des durées d'exécution respectant les contraintes, mais que parfois les temps d'exécution obtenus dépassait largement les contraintes. L'algorithme aléatoire peut donc générer de bons comme de mauvais plans d'ordonnancement.

Puis nous avons les deux algorithmes qui nous permettent de rencontrer nos contraintes de temps : le Glouton et celui qui est basée sur des listes de priorités. Tous leurs facteurs sont similaires, en effet le Prio a un facteur d'inefficacité de seulement 1.039. Ils ont donc un facteur d'utilisation équivalent et plus élevés que les autres par une marge de 40 % (sauf pour l'aléatoire). Ils produisent surtout des ordonnancements ayant le même impact sur les unités de calcul. Ainsi nous pouvons considérer ces deux algorithmes comme les meilleurs compromis entre durée d'exécution et utilisation des processeurs pour notre type d'application.

Ajoutons que les autres algorithmes qui demandent plus de temps d'ordonnancement comme DMDA, de vol et HEFT ont échoués à atteindre les objectifs dus à leur implémentation. HEFT est à la base utilisée pour des systèmes hétérogènes, comme expliqué dans le chapitre de revue de littérature, il calcule sur quels cœurs il serait plus adéquat d'exécuter une tâche, mais dans le cas d'un système homogène cela n'est pas nécessaire. Dans le cas de l'algorithme de vol, il y a l'ajout de temps d'ordonnancement, mais ses performances décevantes sont principalement dues au mécanisme de vol. Il ne prend pas en compte les dépendances de tâches et il ajoute de la communication entre les cœurs. De plus, le DMDA ajoute du traitement au niveau des demandes en accès mémoire, dans notre cas de test, il n'était forcément pas nécessaire dû à la localité des données, mais dans un cas où nous aurions plus d'unités de calculs réparties sur de multiples cœurs, il pourrait être plus efficace que les autres algorithmes d'ordonnancement traités à travers ce mémoire.

Le dernier point à étudier à partir de ces résultats est l'impact du nombre de cœurs sur la gestion des tâches. Frigon et al [42] émettaient l'hypothèse qu'afin d'avoir une virtualisation adéquate

d’un lien LTE montant, il fallait dédier une unité de calcul complète à la gestion des tâches. Nous observons à travers nos résultats que lorsque le pourcentage d’ordonnancement augmente sur un cœur, nous ne pouvons pas respecter nos contraintes. De plus nous observons que retirer un cœur pour l’exécution de tâches, comme vu avec l’algorithme d’ordonnancement DMDA, augmente notre facteur d’inefficacité. Ces conséquences néfastes sont surtout dues à la perte d’unité de calcul. Moins de cœurs de calculs impliquent une utilisation accrue des autres unités de calculs. Ainsi nous en déduisons qu’avoir les fonctions des algorithmes d’ordonnancements distribués sur plusieurs cœurs à la fois est plus efficace qu’avoir un cœur dédié et est aussi plus flexible, car permet de faire des changements au niveau même de l’unité de calcul , réduisant donc les accès mémoire.

CHAPITRE 6 CONCLUSION ET RECOMMANDATIONS

Ce chapitre résume les contributions du mémoire tout en identifiant ses limites et ses contraintes. Il nous permet également d'identifier de nouvelles voies de recherche, ainsi que de formuler des recommandations pour le futur.

6.1 Sommaire des travaux et contributions

À travers ce mémoire, nous avons étudié de nombreux concepts. Au cours de ces travaux de recherche, la littérature a d'abord été parcourue pour cerner les enjeux et tendances dans le monde de la virtualisation des protocoles de communications. Grâce à une revue de littérature assez importante, nous avons pu étudier une tendance lourde vers la création d'une plateforme adaptée pour la virtualisation de protocoles de télécommunication sous la forme d'un C-RAN. En effet, nous avons démontré de manière analytique qu'une grappe de calcul utilisant du matériel COTS (Commercial Off The Shelf) possède une latence assez basse et une bande passante assez élevée pour traiter les services présents et futurs compris dans le protocole LTE.

De plus, nous avons pu étudier les avantages et inconvénients associés à l'algorithme d'ordonnancement au sein d'un système d'exécution. Ce dernier possède tout le contrôle sur l'efficacité de traitement des tâches du système. De par ce fait, le code qui est implémenté a un gros impact sur le rendement de l'application. Après expérimentations et tests, nous avons pu démontrer que :

- La gestion des tâches associées à la virtualisation d'un lien LTE ascendant est faisable sur une plateforme contenant un processeur i7 d'Intel sous un système d'exécution StarPU.
- Plusieurs algorithmes d'ordonnancement seraient adéquats dans les contextes mis en place. En effet, des algorithmes comme le glouton offrent un bon compromis entre vitesse d'exécution et pourcentage d'utilisation des unités de calculs.
- La gestion des tâches n'a pas besoin d'avoir un processeur dédié. La spécialisation d'un processeur induit la perte d'une unité de calcul pour l'exécution des tâches, résultant en une performance moindre. Répartir les fonctions d'ordonnancement sur plusieurs cœurs à

la fois est donc plus efficace et flexible, car cela permet de faire des changements au niveau même de l'unité de calcul, réduisant donc les accès mémoire.

6.2 Limitations de la solution proposée

La première chose à considérer est l'environnement dans lequel les tests ont été réalisés. En effet, tel que nous l'avons expliqué dans le plan de test, nous utilisons une machine virtuelle. Bien que cela ait été fait afin de pouvoir utiliser une version Linux qui permettrait plus de contrôle sur les unités de calculs, l'ajout de l'interface fournie par Virtual box ajoute des aléas au sein de l'exécution des tâches sur les cœurs. La couche supplémentaire peut rajouter du temps aux traitements de tâches en utilisant de la puissance de calcul des cœurs, mais surtout cela induit plus d'interruptions affectant chaque tâche devant être exécutée. Il faut passer à travers deux systèmes d'exploitation, Linux puis Windows pour traiter les interruptions, à cause de l'utilisation d'une machine virtuelle. On pourrait remédier à la situation en utilisant une machine ayant une version allégée d'un Linux dédié.

Un autre problème que nous pouvons attribuer aux systèmes d'exploitation de la plateforme utilisée est le manque de contrôle sur les niveaux d'antémémoires des processeurs. Non seulement le système d'exploitation peut, d'une manière qui pourrait nous sembler aléatoire, accéder à l'antémémoire de façon prioritaire, mais il peut aussi interrompre des accès et invalider des zones d'antémémoires préalablement mises en place. Cela est dû au fait que le mécanisme de « cache locking » n'existe pas sur la présente architecture de microprocesseurs, et ce phénomène ne pourra malheureusement pas être remédié dans la courante génération de processeurs.

En ce qui a trait aux limitations apportées par l'implémentation du code testé, nous devons noter que l'ajout des outils de profilage ralentit l'exécution de l'application, rendant donc les résultats tous moins efficaces que dans le cas réel, et ce par une marge équivalente entre chaque algorithme. Enfin, ajoutons que la performance des algorithmes plus complexes tels que HEFT et DMDA est directement reliée aux nombres d'unités de calculs disponibles. En effet, plus il y a de cœurs et plus ces algorithme prévalent et sont efficaces. Lors de l'implémentation d'une station de base virtuelle, il ne faudrait donc pas omettre d'utiliser ces algorithmes, car ils pourraient s'avérer plus efficaces dans un contexte plus large.

Cependant, plusieurs points pourraient être améliorés :

- Nous pourrions perfectionner les modèles de performance utilisés à travers les tests. Cela nous permettrait d'avoir de meilleurs résultats à travers les différents algorithmes.
- Nous pourrions aussi utiliser des processeurs avec plus d'unités de calculs, car nous étions limités à 3 doubles filaires, donc 6 cœurs de calculs, à cause de l'utilisation d'une machine virtuelle.
- Nous aurions pu aussi implémenter un profil de tâche sous la spécification de tâches parallèles. Il s'agit d'une technique de programmation permettant au système d'exécution de savoir « d'avance » que le profil de tâches devant être exécuté sera hautement parallèle. Un tel type de programmation favorise les algorithmes d'ordonnancement comme HEFT et DMDA, mais n'ajoute que peu d'avantages lors de l'utilisation d'algorithmes plus simple comme le glouton.

6.3 Travaux futurs

Les tests effectués au sein de mes travaux de recherche se voulaient une preuve de concept. Les travaux futurs visent à améliorer les connaissances associées aux résultats obtenus et surtout à aller de l'avant dans l'étude d'une grappe de calcul temps réel.

Pour pallier au fait que les tests n'utilisent qu'une émulation du vrai profil de tâches, il serait pertinent de refaire ces mêmes tests avec les vraies tâches produites lors d'une virtualisation d'une station de base LTE. Il faudrait donc intégrer la bibliothèque de calcul MKL et utiliser plus d'extension C afin d'avoir un code le plus efficace possible. De plus, compte tenu des résultats obtenus, il serait bien d'effectuer les tests sur une vraie grappe de calcul. En effet, bien que nous ayons adéquatement étudié les latences et bandes passantes associées avec la plateforme, nous pourrions avoir des interférences indésirables lors des communications inter-serveurs. Les résultats sortants de ces tests nous permettraient d'évaluer l'importance d'accélérateurs matériels et ainsi de valider l'importance d'une plateforme telle que celle décrite dans le chapitre 3.

Le dernier point que nous devrons explorer est l'identification d'une manière de rendre ce type de plateforme plus accessible au « grand public », c'est-à-dire rendre de rendre les implémentations

d’accélérateurs matériels plus facilement exploitables pour les utilisateurs de serveurs. Notre tâche sera facilitée lors de l’arrivée des nouveaux processeurs d’Intel avec une architecture Skylake. Il est en effet prévu que ces derniers aient un FPGA sur puce intégrée. Ainsi, il ne sera pas nécessaire d’avoir du matériel supplémentaire, il suffira de mettre les processeurs dans les socles correspondants et le tour sera joué. Ajoutons aussi qu’il est prévu que cette microplaquette offre un mécanisme de « cache locking » intégré, nous permettant d’avoir des communications et un système d’antémémoire plus déterministe.

BIBLIOGRAPHIE

[1] Y. Lin, L. Shao, Z. Zhu, Q. Wang, and R. K. Sabhikhi, "Wireless network cloud: Architecture and system requirements," *IBM J. Res. Dev.*, vol. 54, no. 1, pp. 4:1–4:12, Jan. 2010.

[2] S. Bhaumik, S. P. Chandrabose, M. K. Jataprolu, G. Kumar, A. Muralidhar, P. Polakos, V. Srinivasan, T. Woo, and U. C. Berkeley, "CloudIQ : A Framework for Processing Base Stations in a Data Center," in *ACM MobiCom 2012*, 2012, pp. 125–136.

[3] A. Vajda, *Multi-core and Many-core Processor Architectures*. 2011, pp. 9–43.

[4] R. N. Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault, "Data-Aware Task Scheduling on Multi-Accelerator based Platforms.," in *16th International Conference on Parallel and Distributed Systems*, 2010.

[5] "Task-Based Programming." [Online]. Available: https://software.intel.com/en-us/node/506100. [Accessed: 18-Jan-2015].

[6] H. Topcuoglu and S. Hariri, "Task scheduling algorithms for heterogeneous processors," in *Proceedings. Eighth Heterogeneous Computing Workshop (HCW'99)*, 1999, pp. 3–14.

[7] H. Topcuoglu and S. Hariri, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.

[8] China Mobile, "C-RAN The Road Towards Green RAN," 2013.

[9] P. Sen Margaret Chiosi, Don Clarke, Peter Willis, Andy Reid,James Feger, Michael Bugenhagen, Waqar Khan, Michael Fargano, Dr. Chunfeng Cui, Dr. Hui Deng,Javier Benitez,Uwe Michel, Herbert,Masaki Fukui, Katsuhiro Shimano, Damker,Kenichi Ogaki, Tetsuro Matsuzaki,D, "Network Functions Virtualisation," in *SDN and OpenFlow World Congress*, 2012, no. 1, pp. 1–16.

[10] L. E. Li, Z. M. Mao, and J. Rexford, "Toward Software-Defined Cellular Networks," *2012 Eur. Work. Softw. Defin. Netw.*, pp. 7–12, Oct. 2012.

[11] E. Dahlman, S. Parkvall, and J. Skold, *4G: LTE/LTE-Advanced for Mobile Broadband*, Second. Elsevier Ltd, 2014, p. 455.

[12] Freescale Semiconductor, "Long Term Evolution Protocol Overview," 2008.

[13] S. Saini, H. Jin, D. Jespersen, H. Feng, J. Djomehri, W. Arasin, R. Hood, P. Mehrotra, and R. Biswas, "An early performance evaluation of many integrated core architecture based SGI rackable computing system," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*, 2013, pp. 1–12.

[14] S. Thibault and R. Namyst, "StarPU : a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines StarPU : a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines," 2010.

[15] J. Agron, D. Andrews, M. Finley, W. Peck, and E. Komp, "FPGA Implementation of a Priority Scheduler Module."

[16] J. Teller, F. Özgüner, and R. Ewing, "Scheduling Task Graphs on Heterogeneous Multiprocessors with Reconfigurable Hardware," in *2008 International Conference on Parallel Processing - Workshops*, 2008, pp. 17–24.

[17] T. Hagras and J. Janeček, "Static vs . Dynamic List-Scheduling Performance Comparison," *Acta Polytech.*, vol. 43, no. 6, pp. 16–21, 2003.

[18] N. Rajak and A. Dixit, "Classification of List Task Scheduling Algorithms : A Short Review Paper," *J. Ind. Intell. Inf.*, vol. 2, no. 4, pp. 320–323, 2014.

[19] S. L. Stanzani, L. M. Sato, and M. A. S. Netto, "Scheduling Workflows in Multi-cluster Environments," in *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, 2013, pp. 560–565.

[20] D. M. Abdelkader and F. Omara, "Dynamic task scheduling algorithm with load balancing for heterogeneous computing system," *Egypt. Informatics J.*, vol. 13, no. 2, pp. 135–145, Jul. 2012.

[21] S. Saini, H. Jin, R. Hood, D. Barker, P. Mehrotra, and R. Biswas, "The impact of hyper-threading on processor resource utilization in production applications," in *2011 18th International Conference on High Performance Computing*, 2011, pp. 1–10.

[22] S. Jin, G. Schiavone, and D. Turgut, "A performance study of multiprocessor task scheduling algorithms," *J. Supercomput.*, vol. 43, no. 1, pp. 77–97, Jun. 2007.

[23] B. Cirou and E. Jeannot, "Triplet: A clustering scheduling algorithm for heterogeneous systems," in *Proceedings International Conference on Parallel Processing Workshops*, 2001, pp. 231–236.

[24] T. Yang, A. Gerasoulis, and N. Brunswick, "DSC : Scheduling Parallel Tasks on an Unbounded Number of Processors," in *Proceedings of Supercomputing 91*, 1993, no. 1, pp. 1–33.

[25] G. C. Buttazzo, “Rate Monotonic vs. EDF: Judgment Day,” *Real-Time Syst.*, vol. 29, no. 1, pp. 5–26, Jan. 2005.

[26] M. Vetromille, L. Ost, C. A. M. Marcon, C. Reif, and F. Hessel, “RTOS Scheduler Implementation in Hardware and Software for Real Time Applications,” in *Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP’06)*, 2006, pp. 163–168.

[27] R. Mancuso, P. Srivastava, D. Chen, and M. Caccamo, “A Hardware Architecture to Deploy Complex Multiprocessor Scheduling Algorithms,” in *IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2014, pp. 1–10.

[28] A. Cutler and E. M. Services, “More-than-Moore.” 2008.

[29] J. Reinders, “An Overview of Programming for Intel Xeon processors and Intel Xeon Phi™ coprocessors,” 2012.

[30] J. Stuecheli, “Power Technology For a Smarter Future Power Technology For a Smarter Future The Best Time for Power,” 2014.

[31] C. Analytics, “Infrastructure Matters: POWER 8-based Power Systems vs. x86 Servers,” 2014.

[32] I. Corporation, “Intel® Threading Building Blocks User Guide.” [Online]. Available: https://software.intel.com/fr-fr/node/506045. [Accessed: 18-Jan-2015].

[33] Intel Corporation, “Cilk Documentation - CilkPlus.” [Online]. Available: https://www.cilkplus.org/cilk-documentation-full. [Accessed: 18-Jan-2015].

[34] H. Li, “A Cilk Implementation of LTE Base-station up- link on the TILEPro64 Processor,” Chalmers University of Technology, 2011.

[35] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, “StarPU : a unified platform for task scheduling on heterogeneous multicore architectures,” *Concurr. Comput. Pract. Exp.*, vol. 23, no. 2, pp. 187–198, 2010.

[36] C. S. and A. I. L. at MIT, “StreamIt.” [Online]. Available: http://groups.csail.mit.edu/cag/streamit/. [Accessed: 18-Jan-2015].

[37] ETSI Group, “- Network Functions Virtualisation (NFV); Use Cases,” 2013.

[38] A. De Domenico, “Final report on MAC/RRM state-of-the-art, Requirements, scenarios and interfaces in the iJOIN architecture,” 2013.

[39] U. Salim, “State-of-the-art of and promising candidates for PHY layer approaches on access and backhaul network,” 2013.

[40] W. Paper, “C-RAN The Road Towards Green RAN,” vol. 5, 2011.

[41] J. Huang, R. A. N. Duan, C. Cui, and J. X. Jiang, “Recent Progress on C-RAN Centralization and Cloudification,” *IEEE Access*, vol. 2, 2014.

[42] Huawei and C. Co, “Latency and jitter characterization of wireless communication protocols on blade servers,” 2014.

[43] A. B. Labs, “Architecture options for evolved C-RAN Lessons learned,” 2013.

[44] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta, “A reconfigurable computing system based on a cache-coherent fabric,” *Proc. - 2011 Int. Conf. Reconfigurable Comput. FPGAs, ReConFig 2011*, pp. 80–85, 2011.

[45] A. Ericsson AB, Huawei Technologies Co. Ltd, NEC Corporation, Nortel Networks Ltd and L. and N. S. N. G. & C. KG, “CPRI Specification,” 2009.

[46] C. Lanzani and S. P. Manager, “Open Base Station Architecture,” 2008.

[47] “Reduce The Cost Of Your CPRI Links For 3G And 4G Wireless.” [Online]. Available: http://electronicdesign.com/4g/reduce-cost-your-cpri-links-3g-and-4g-wireless. [Accessed: 15-Feb-2015].

[48] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek, “Intel® QuickPath Interconnect Architectural Features Supporting Scalable System Architectures,” in *High Performance Interconnects (HOTI)*, 2010, pp. 1–6.

[49] P. SIG, “PCI Express Base 3.0 Specification,” 2015. [Online]. Available: https://www.pcisig.com/specifications/pciexpress/base3/. [Accessed: 27-Jan-2015].

[50] J. Gong, T. Wang, J. Chen, H. Wu, F. Ye, S. Lu, and J. Cong, “An Efficient and Flexible Host-FPGA PCIe Communication Library,” in *24th International Conference on Field Programmable Logic and Applications*, 2014, pp. 1–6.

[51] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, “Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System,” in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009, pp. 261–270.

[52] “QPI Interface Solution.” [Online]. Available: http://www.xilinx.com/applications/data-center/servers/server_design_example.html. [Accessed: 02-Dec-2014].

[53] Xilinx Corporation, “7 Series FPGAs GTX / GTH Transceivers User Guide,” 2012.

[54] Xilinx Corporation, “Virtex ® UltraScale ™ FPGAs,” 2014.

[55] Intel Corporation, “Intel® Xeon® Processor E5-2670(20M Cache, 2.60 GHz, 8.00 GT/s Intel® QPI),” 2012. [Online]. Available: http://ark.intel.com/fr/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI. [Accessed: 28-Jan-2015].

[56] Xilinx Corporation, “Xilinx Virtex-7 FPGA VC709 Connectivity Kit.” [Online]. Available: http://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html. [Accessed: 29-Jan-2015].

[57] ASUS Corporation, “Motherboards - Z9PE-D8 WS - ASUS,” 2014. [Online]. Available: http://www.asus.com/Motherboards/Z9PED8_WS/. [Accessed: 29-Jan-2015].

[58] Xilinx Corporation, “10 Gigabit Ethernet MAC v14.0,” 2000.

[59] Xilinx Corporation, “10 Gigabit Ethernet PCS/PMA v5.0,” 2000.

[60] L. Wen, J. Wang, and J. Sun, “Detectinng Implicit Dependencies Between Tasks from Event Logs,” pp. 591–603, 2006.

[61] “StarPU : A Unified Runtime System for Heterogeneous Multicore Architecture.” [Online]. Available: http://starpu.gforge.inria.fr/. [Accessed: 09-Apr-2015].