Real-time Software Architectures and Performance Evaluation Methods for 5G Radio Systems

Tsu-Han Wang

▶ To cite this version:

Tsu-Han Wang. Real-time Software Architectures and Performance Evaluation Methods for 5G Radio Systems. Networking and Internet Architecture [cs.NI]. Sorbonne Université, 2022. English. NNT : 2022SORUS362 . tel-04343687

HAL Id: tel-04343687
https://theses.hal.science/tel-04343687

Submitted on 14 Dec 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

SORBONNE UNIVERSITÉ

EURECOM
Sophia Antipolis

Real-time Software Architectures and Performance Evaluation Methods for 5G Radio Systems

Dissertation

submitted to

Sorbonne Université

in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Author:

Tsu-Han WANG

Scheduled for defense on the 13th December, 2022, before a committee composed of:

Reviewers

Dr.	**Thierry Turletti**	INRIA, France
Prof.	**Ludovic Apvrille**	Télécom Paris, France

Examiners

Prof.	**Melek Önen**	EURECOM, France
Prof.	**Ternying Hsu**	NCTU, Taiwan

Director of Thesis

Prof.	**Raymond KNOPP**	EURECOM, France

Architectures logicielles temps-réel et m'éthodes d'évaluation des performances les systèmes radio 5G

Thèse

soumise à

Sorbonne Université

pour l'obtention du Grade de Docteur

Auteur:

Tsu-Han WANG

Soutenance de thèse effectuée le 13 Decembre 2022 devant le jury composé de:

Rapporteurs

Dr.	**Thierry Turletti**	INRIA, France
Prof.	**Ludovic Apvrille**	Télécom Paris, France

Examinateurs

Prof.	**Melek Önen**	EURECOM, France
Prof.	**Ternying Hsu**	NCTU, Taiwan

Directeur de Thèse

Prof.	**Raymond KNOPP**	EURECOM, France

To my family

Abstract

Cette thèse a pour sujet l'optimisation de la radio logicielle (SDR, software defined radio) pour la 5G. Le premier point à aborder est la radio logicielle. Comme son nom l'indique, la radio logicielle est un programme informatique qui est exécuté par un ordinateur se comportant comme une station radio. Ce qui rend ce système particulier c'est qu'il sagit d'un logiciel, avec toutes les caractéristiques d'un logiciel, mais qui exécute ce qui jusqu'alors était réservé au matériel. Plusieurs raisons expliquent la popularité croissante de la radio logicielle. L'une d'elles est que la conception d'une puce électronique est longue et tolère peu les erreurs. Pour produire une puce électronique, il faut tout d'abord passer par une étape de conception utilisant un langage de bas niveau de type Verilog pour décrire le comportement du système. Ensuite ce programme doit être traduit en un graphique où la taille de chaque porte logique, comment elle est implémentée, et où elle est positionnée, sont importants puisque la taille de la puce a un impact direct sur son coût de fabrication. De même, les connections entre les portes de la puce doivent être soigneusement conçues puisque les longs chemins induisent des délais dans l'acheminement des données ce qui pourrait générer des erreurs. Et il ne s'agit là que d'un coup d'œil sur la conception de la puce dans la phase graphique. Après toutes ces étapes, ce graphique va à l'usine où la gravure sur puce réelle est effectuée. Il faut traverser le revêtement, graver la couche pour révéler le silicium, graver le silicium, et laver les résidus. Ces étapes ne servent à produire qu'une seule couche de la puce, qui en contient plusieurs. Cela montre bien qu'il faut du temps et de la précision pour fabriquer une puce. De tout ce qui précède, on peut dire que produire une puce est à la fois coûteux et long.

La radio logicielle bénéficie des avantages du développement logiciel, avec un cycle de vie de développement court et une facilité certaine pour ajouter de nouvelles fonctionalités. Cependant, des obstacles demeurent que la radio logicielle doit surmonter, le premier étant le temps réel. Les puces matérielles nécessitent toute cette complexité de conception pour s'assurer qu'elles peuvent traiter le signal de la manière la plus rapide possible. Les programmes informatiques qui tournent sur des ordinateurs à usage général ne peuvent pas égaler les performances des puces matérielle. La bonne nouvelle pour la radio logicielle est que les processeurs sont de plus en plus puissants avec les machines récentes donc même si les logiciels ne peuvent rivaliser en terme de performance avec le matériel, ils peuvent quand même gérer les procédures dans le temps imparti. De plus, malgré cet inconvénient de moindre performance, le logiciel a l'avantage évident d'avoir un cycle de développement court. Dès que de nouvelles fonctionalités doivent être ajoutées à la radio logicielle, il suffit simplement d'introduire une nouvelle fonction qui exécute cette

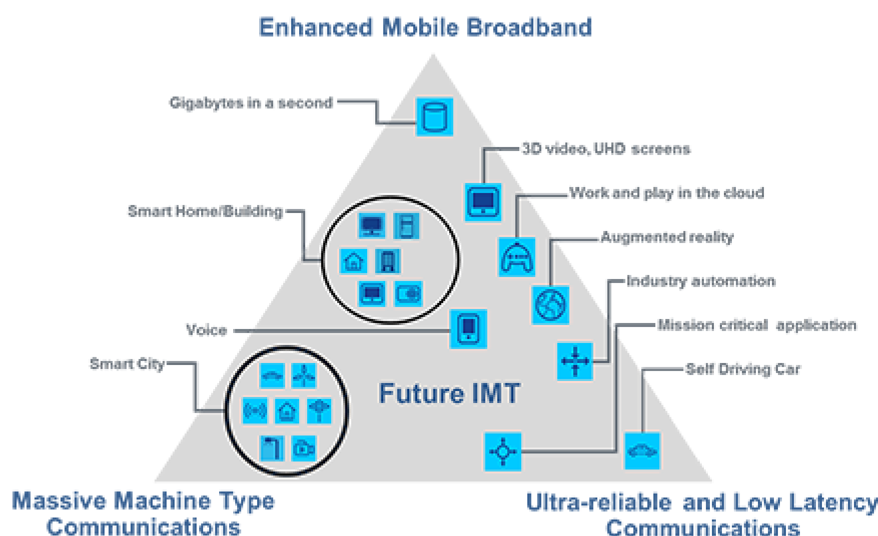

Figure 3 – IMT-2020

fonctionalité et de recompiler le programme. Et aussi, il peut prendre en charge plusieurs configurations car pour le programme, il ne s'agit que d'un ensemble de variables qui doivent être prédéfinies par la configuration. Donc au final, de plus en plus de gens s'intéressent à la radio logicielle en raison de sa souplesse de développement et tolèrent sa lenteur puisque les traitements sont quand même faits dans les temps.

Le deuxième point à aborder est la 5G. La 5G a trois caractéristiques principales qui sont le haut débit mobile amélioré (Enhanced Mobile Broadband, eMBB), les communications ultra-fiables et à faible latence (Ultra-reliable and Low Latency Communications, URLLC), et les communications massives de type machine (Massive Machine Type Communications, mMTC), comme le montre la figure 3. L'eMBB se concentre sur les débits de données massifs avec une large bande où la transmission de données est la principale préoccupation tandis que l'URLLC, de par son nom, se concentre d'avantage sur la précision et les réponses rapides pour des usages avec un besoin de rétroaction rapide, la conduite automatique par exemple. Quant à mMTC, il s'agit plus de gérer simultanément une quantité massive d'appareils sur une large portée où la quantité d'appareils gérés en même temps est la principale préocuppation, par exemple l'internet des objets (Internet of Things, IoT)

Ce qui précède decrit ce qu'est la radio logicielle 5G. Ce dont va traiter cette thèse est la couche physique (PHY). Il y a sept couches pour la connection internet et la couche physique est la plus basse de toutes. Ce que fait la couche physique c'est principalement de gérer le flux binaire et de transmettre et recevoir ce flux entre les appareils. Pour pouvoir parler de communication sans fil nous devons tout d'abord parler d'une des plus importantes conceptions dans le domaine des communications, le multiplexage par répartition orthogonale de la fréquence (Orthogonal Frequency-Division Multiplexing, OFDM). Le principe d'OFDM est d'avoir plusieurs porteuses dans le domaine fréquentiel

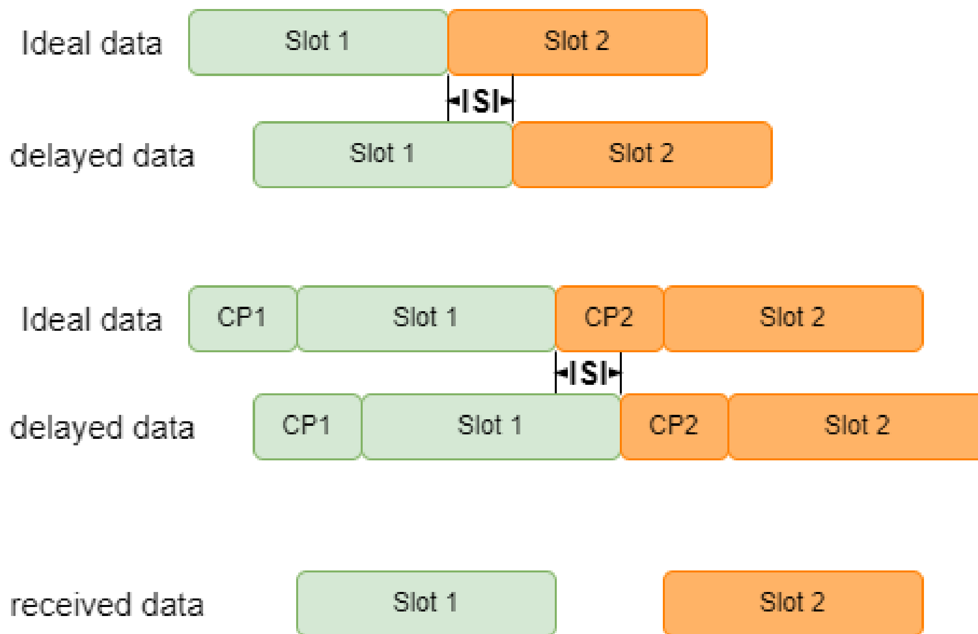

Figure 4 – Inter Symbol Interference (ISI)

orthogonales pour transmettre les données. Ainsi, idéalement, il y aura des interférences seulement où ne se trouvent pas les données. Cependant en pratique, il existe toujours des interférences inter-symboles (Inter-Symbol Interference, ISI) et des interférences inter-porteuses (Inter-carrier Interference, ICI). ISI et ICI se produisent à cause du délai d'arrivée du paquet ce qui signifie qu'il y aura des données manquantes et/ou melangées dans le signal reçu. ISI peut se résoudre en ajoutant une période de garde contenant des données connues, par exemple du remplissage par des zéros, comme illustré par la figure 4. Bien que l'ajout de données connues résolve le problème ISI, cela ne résout pas l'ICI. L'introduction du préfixe cyclique (Cyclic Prefix, CP) au lieu d'un remplissage par des données connues résout le problème de l'ICI puisque le décalage temporel dans le domaine temporel n'est qu'un décalage dans le domaine fréquentiel, comme illustré à la figure 5, les données sont toujours récupérables.

Étant donné que tout le travail de la procédure PHY vise à ce que le signal OFDM soit transféré et décodé avec succès même face à des interférences lors du passage dans le canal, ce dont PHY doit se préoccuper, c'est de savoir comment obtenir des données de flux binaire réversibles avec une tolérance pour les bruits. Prenons l'exemple de la transmission descendante (downlink). Les données de flux binaire proviennent de la couche supérieure et subissent ensuite un traitement au niveau du bit qui consiste en un brouillage et un codage. Le brouillage modifie l'ordre des données du flux binaire pour tenir compte des cas d'interférences en rafale qui perturbent les données consécutives. Quant au codage, il consiste à ajouter des bits de contrôle selon un algorithme de codage donné pour s'assurer que même si certains bits sont contaminés par le bruit, ils sont toujours récupérables. Et après le brouillage et le codage, vient la modulation

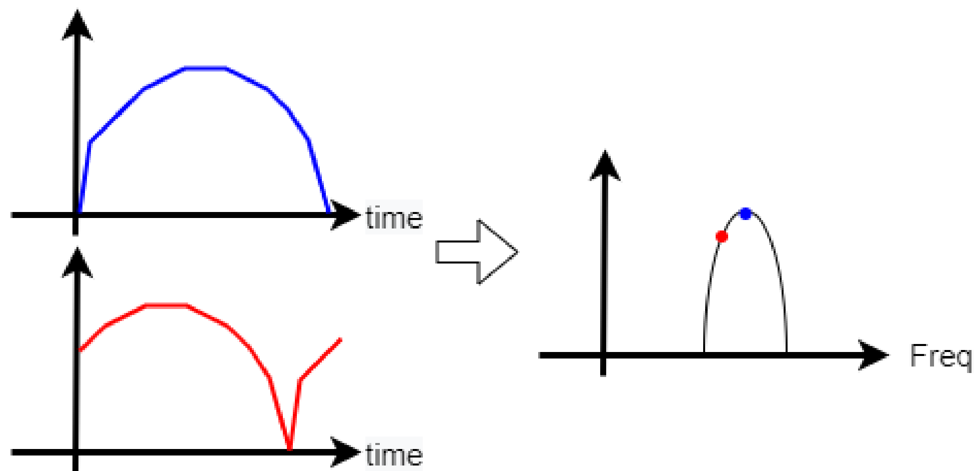

Figure 5 – Inter Carrier Interference (ICI)

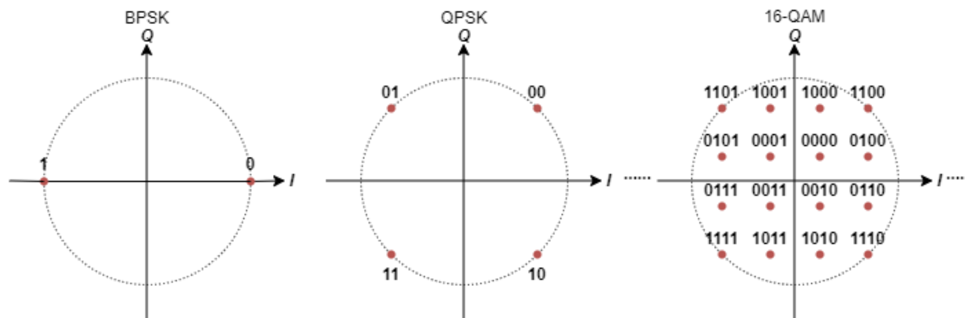

Figure 6 – Quadrature Amplitude Modulation (QAM) diagram example

d'amplitude en quadrature (Quadrature Amplitude Modulation, QAM). Ce que fait QAM, c'est mapper les données de flux binaires sur des ensembles d'amplitudes et d'angles afin qu'ils puissent être ajoutés à l'onde OFDM, comme illustré à la figure 6. Enfin il y a le mappage OFDM. Ce n'est que la procédure pour une couche et une antenne. Pour la 5G, il est nécessaire de disposer de plusieurs antennes pour prendre en charge les entrées multiples et les sorties multiples (Multiple-Input Multiple-Output, MIMO). Pour le fonctionnement MIMO, le mappage de plusieurs couches sur plusieurs antennes est effectué par mappage et précodage de couche pour s'assurer que chaque antenne a le signal qu'elle doit envoyer, puis on effectue la transformée de Fourier rapide inverse (Inverse Fast Fourier Transform, IFFT) pour répondre aux exigences OFDM.

Comme mentionné précédemment, tous les processus doivent être faits « à temps », ce qui rend la contrainte de temps de plus en plus importante. Dans la norme 3GPP pour la 5G, la relation entre l'espacement des sous-porteuses et le slot est indiquée dans le tableau 1

Les équations suivantes montrent la relation entre l'espacement des sous-porteuses

μ	$\Delta f = 2^{\mu} * 15[kHz]$	Cyclic prefix
0	15	Normal
1	30	Normal
2	60	Normal, Extended
3	120	Normal
4	240	Normal

Table 1 – Numérologie et espacement des sous-porteuses dans la norme 3GPP

et le slot où le slot est l'unité de temps de base dans PHY. $T_c = 1/(\Delta f_{max} * N_f)$ où $\Delta f_{max} = 480 * 10^3$HZ et $N_f = 4092$, La constante $k = T_s/T_c = 64$ où $T_s = 1/(\Delta f_{ref} * N_{f,ref})$,$\Delta f_{ref} = 15 * 10^3$HZ et $N_{f,,ref} = 2048 T_f = (\Delta f_{max} N_f / 100) * T_c = 10ms$. Ces équations montrent que le temps de trame est toujours de 10 ms, mais qu'il y aura plus de slots dans une trame lorsque la numérologie augmentera, ce qui signifie que le slot devient plus court tout en utilisant une numérologie plus grande. La numérologie zéro est la même configuration que celle utilisée par LTE où le slot est de 1 ms, ce qui signifie que le slot en 5G est généralement plus court que ce qui est demandé en LTE.

Connaissant la durée de l'unité de base, le slot, dans PHY, nous savons maintenant que le temps d'exécution de toutes les procédures du processus PHY, une fois tout additionné, doit être compris dans un slot pour que la radio logicielle s'exécute de manière stable. Il est alors important de savoir combien de temps coûte chaque fonction individuelle et combien de temps coûte l'ensemble de la procédure. Parlons d'exécution séquentielle. Ce qui est fait est de programmer la fonction PHY directement à partir de la norme 3GPP avec le même ordre de flux d'exécution, comme illustré à la figure 7. Et puisque le canal répond quatre slots plus tard dans le canal opposé, comme indiqué à la figure 8, on a instinctivement le fonctionnement suivant : traiter le slot n pour un côté (transmission ou réception), puis le côté opposé pour le slot n+4. Il est plus logique d'avoir le système piloté par la réception que par l'émission puisque le système doit activer la procédure de réception à chaque tranche de temps et ce n'est qu'à ce moment-là que les données sont connues pour la transmission. Pourtant, la transmission peut être pré-générée bien avant qu'elle ne soit transmise. Ce qui aboutit finalement à la structure décrite à la figure 9.

D'après la figure 9, on peut voir que la transmission est préparée bien plus tôt que ce qui est nécessaire, où le temps total d'exécution pour une réception et une transmission est preque le même que la durée d'un slot dans LTE quand il y a une forte charge. Avec un ordonnancement si serré, la préparation précoce des données transmises est considérée comme un gaspillage d'énergie et de ressources. Sans oublier qu'avoir un ordonnancement plus serré pour des slots plus courts en 5G rend cette structure peu pratique, que l'accélération doit améliorer. Il existe deux idées principales pour l'accélération. La première est de réorganiser la structure ce qui signifie avoir de multiples exécutions parallèles en même temps, par exemple « time space trading ». La deuxième est d'accélérer la fonction elle-même.

Premièrement, parlons de la réorganisation de la structure. Il est important de savoir que tous les canaux ont peu de relations les uns avec les autres comme illustré par la figure

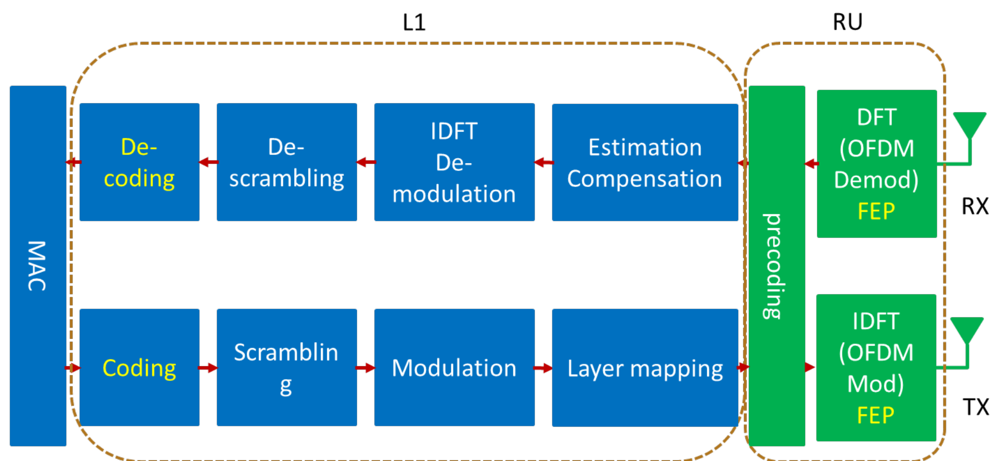

Figure 7 – Flux de procédure PHY pour la radio logicielle

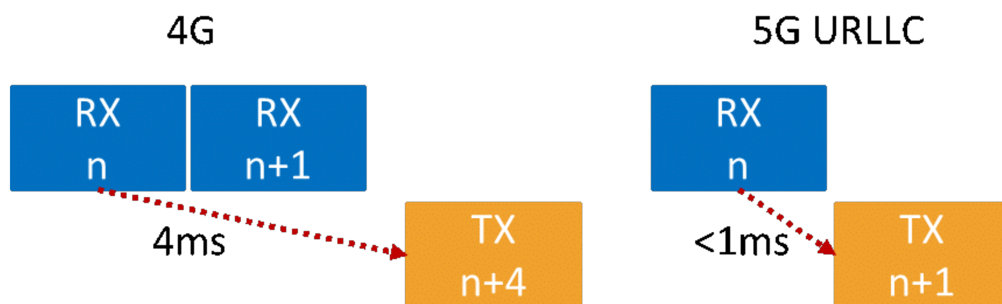

Figure 8 – Réponse du canal dans différents modes

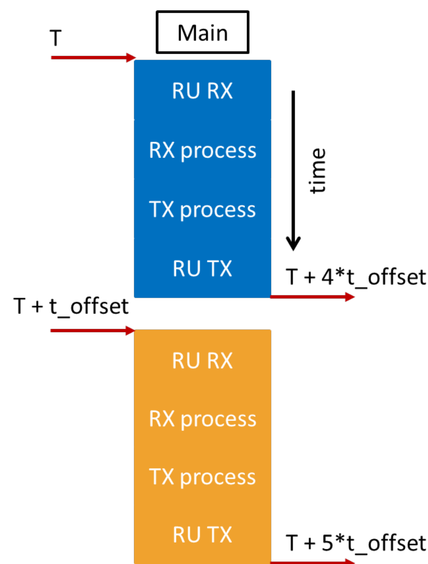

Figure 9 – Procédure pour l'exécution séquentielle radio logicielle

10. Cela signifie que la plupart des canaux peuvent être traités en parallèle. Pourtant, le découpage du traitement des canaux en morceaux est un peu difficile à gérer, donc l'accent sera mis sur ceux qui consomment le plus de temps et de ressources, à savoir les canaux partagés, responsable de la majeure partie du transfert de données. Il y a deux canaux partagés, un pour la liaison montante et un pour la liaison descendante. En plus du traitement du canal, le traitement d'antenne OFDM coûtera également un peu de temps lors de l'utilisation de plusieurs antennes. Ainsi, les quatre principales parties chronophages sont la procédure OFDM pour la réception et la transmission et la procédure de canal partagé pour la réception et la transmission. Étant donné que l'ordre de ces quatre procédures est irréversible, nous devons proposer une structure qui conserve l'ordre tout en les exécutant en même temps. Bien que cela puisse sembler un peu contradictoire, il existe en fait une solution qui s'appelle un pipeline.

La structure de pipeline d'origine est illustrée à la figure 11. La structure de pipeline traditionnelle utilise une horloge globale qui déclenche les étapes pour démarrer le traitement, aux instants marqués par la ligne rouge sur la figure 11, mais il peut y avoir quelques problèmes. Le premier problème que l'on peut rencontrer est de diviser le processus de manière inégale, ce qui est une variation temporelle. Chaque étape du pipeline a son propre temps d'exécution qui est très probablement différent des autres puisque chaque étape a sa propre procédure unique. Si les temps d'exécution de chaque étape sont trop différents les uns des autres, cela signifie que le fractionnement de ce pipeline est déséquilibré, ce qui pourrait non seulement réduire le parallélisme du programme, mais également écraser les données de référence ou les données d'entrée pour l'étape suivante, comme indiqué dans la partie de la variation temporelle de la figure 10. Le deuxième problème est d'avoir un conflit de données. Le conflit de données est un autre problème qui peut survenir lors de l'utilisation de la programmation parallèle.

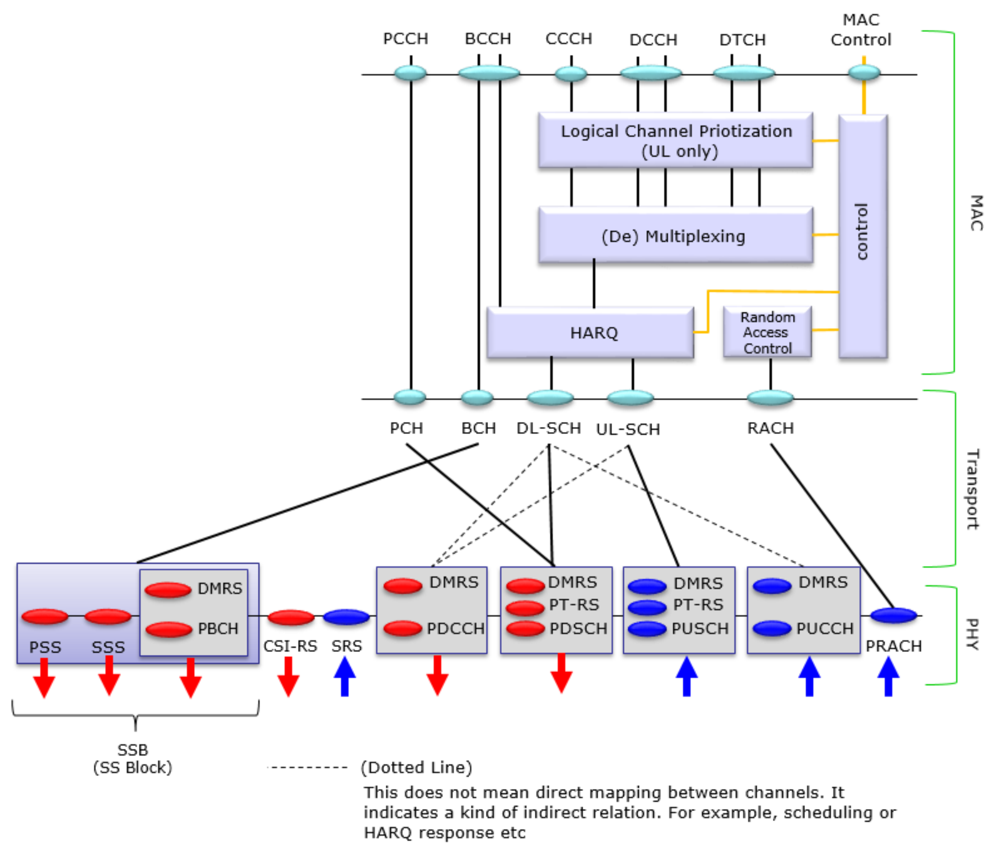

Figure 10 – Mappage des canaux - ShareTechNote

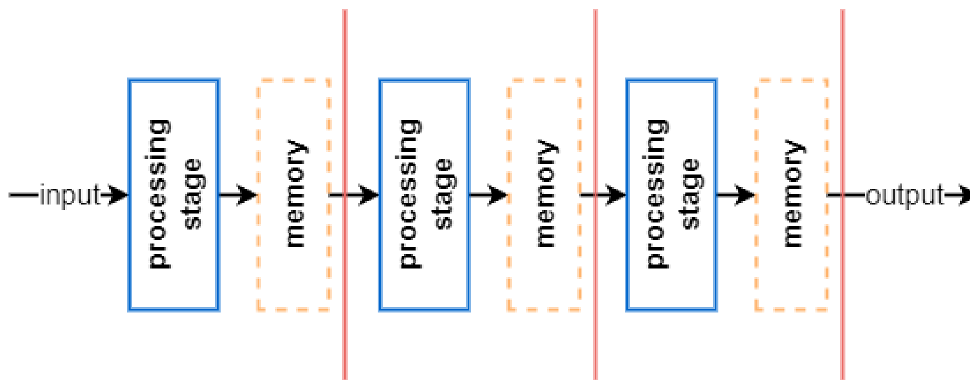

Figure 11 – Un exemple de structure de pipeline à trois étages

Lorsqu'il y a plusieurs écritures qui se produisent en même temps ou lorsqu'une écriture et une lecture se produisent en même temps, l'ambiguïté de qui vient en premier a un impact énorme sur le résultat final. Ce type de risque sur les données rendra le programme instable, ce qui est définitivement indésirable dans la radio logicielle, comme indiqué dans la partie sur les risques sur les données de la figure 12. Le troisième type de problème qui peut survenir dans la programmation parallèle est le risque sur les branches. Si le programme exécute plusieurs branches en parralèle, au moment de la fusion de ces branches pour continuer le processus, il se peut qu'une partie des données ne soit pas encore prête à être gérée en fonction du moment où le processus fusionné est exécuté comme indiqué dans la partie risque sur les branches de la figure 12.

Ici, la structure du pipeline radio logicielle pourrait tirer parti d'une version logicielle où les threads sont responsables du parallélisme. Les threads sont utilisés dans la programmation parallèle car ils peuvent s'exécuter en même temps. En intégrant le pipeline dans une version logicielle, chaque thread représentera une étape de pipeline. Bien que la structure de pipeline d'origine ait été proposée pour la conception matérielle, les problèmes mentionnés précédemment resteront préoccupants après l'intégration de la structure de pipeline dans une version logicielle. Pourtant, le pipelining logiciel présente des avantages, car tous ces problèmes ont une solution plus simple dans une version logicielle où les risques sur les données peuvent être résolus en utilisant le verouillage mutex pour la tâche qui travaille actuellement sur les données. Et les tâches qui voudront également demander l'écriture devront attendre que la mémoire soit libérée puis le système des verrous donnera le droit d'écrire des données à une des tâches en attente, de manière équitable. En ce qui concerne le problème de fusion des branches, le plus important est de s'assurer que la tâche suivante ne commence à s'exécuter qu'une fois toutes les tâches de branche avant la fusion terminées. La solution utilisée dans cette thèse est d'avoir un masque binaire qui contient sa taille en bits qui est au moins de la même taille que le nombre de tâches de branche. Chaque bit du masque représente la branche correspondante et le masque est partagé entre toutes les branches. Une fois que la branche a terminé sa tâche, elle positionnera alors le bit correspondant dans le masque à un pour représenter qu'elle a terminé sa tâche. Chaque branche effectuera la

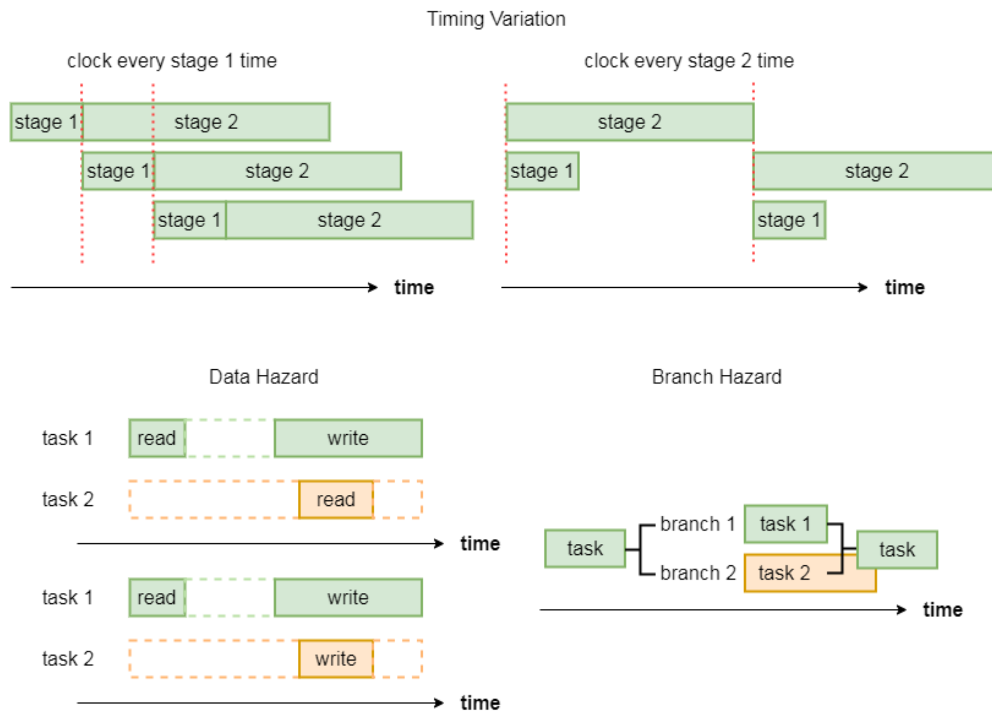

Figure 12 – Risque

vérification du masque et s'il ne reste plus de zéro réveillera la tâche suivante. La même méthode mentionnée ci-dessus pour la prévention des risques sur les données est utilisée pour résoudre les risques sur les branches lors du remplissage du masque. Cela signifie qu'une seule tâche peut remplir le masque à la fois pour s'assurer qu'il n'y aura pas de branches non suivies.

Dans le pipeline logiciel, le déclenchement par l'horloge est remplacé par le signal envoyé. Cela signifie qu'une étape du pipeline logiciel est réveillée par l'étape précédente plutôt que par chaque tic d'horloge. Pour s'assurer que toutes les étapes suivent l'ordre du pipeline, elles seront en mode veille immédiatement après leur création. Après cela, ces étapes vérifient constamment s'il y a un signal de réveil provenant de l'étape précédente. Une fois que chaque étape a terminé son traitement, elle réveille la prochaine étape, puis revient en mode d'attente jusqu'au signal suivant qui la réveillera. Pour pouvoir effectuer le déclenchement, des variables de condition et des mutex sont utilisés pour permettre la communication. Si un thread a besoin d'un autre thread pour le réveiller, il passera à l'état d'attente en verrouillant d'abord la variable de condition, puis en effectuant l'attente sur la variable de condition. Le thread ne reviendra à l'action qu'après la modification de la variable de condition. Le signal correspondant pour réveiller le thread est émis par un autre thread. Quand un thread doit en réveiller un autre, il verrouille la variable de condition et la modifie pour qu'elle corresponde à la condition requise pour l'autre thread, puis envoie le signal afin que l'autre thread sache qu'il est temps de se réveiller et de vérifier la condition. Et puis le thread libère le verrou pour que

d'autres puissent effectuer des opérations sur la même mémoire. À la fin, le thread qui a terminé son travail, et qui aura besoin d'un autre signal pour démarrer un nouveau cycle d'opérations, verrouille alors la variable de condition, modifie à nouveau la variable de condition, déverrouille la variable, puis recommence à attendre.

En combinant toutes ces fonctionnalités avec un pipeline logiciel, à l'exception de la première étape du pipeline, toutes les autres étapes du pipeline doivent passer en état d'attente après la création du thread. Et la première étape du pipeline décide de la fréquence à laquelle ces étapes du pipeline peuvent être déclenchées. La différence est qu'au lieu d'utiliser l'horloge pour permettre à chaque étape du pipeline de démarrer le mouvement suivant, c'est déclenchée uniquement par l'étape précédente, ce qui peut provoquer un brouillage car chaque étape du pipeline a un temps d'exécution différent, et le brouillage se produit généralement sur celui qui prend le plus de temps pour s'exécuter. Pour éviter le brouillage qui était considéré comme l'un des résultats des problèmes de variation temporelle, il suffit simplement de s'assurer que le temps d'exécution de chaque étape ne dépasse pas le temps de la fréquence du déclenchement à partir de la première étape du pipeline. Et la structure du pipeline pour la radio logicielle est illustrée à la figure 13.

Enfin, parlons de l'accélération fonctionelle. Dans la conclusion précédente, les étapes les plus longues sont le canal partagé et la génération de l'OFDM. Il est essentiel d'analyser quelle fonction consomme le plus de temps d'exécution au sein du canal partagé. Comme le montre la figure 5, en prenant la liaison descendante comme exemple, il existe un codage, un brouillage, QAM, un mappage de couche, un précodage et la génération de l'OFDM qui peuvent être la cible de l'accélération.

Dans la 5G, l'algorithme de codage utilise le contrôle de parité à faible densité (Low-Density Parity-Check, LDPC). Ce que fait LDPC, c'est de coder la chaîne de données en attachant une chaîne de bits de contrôle de parité à la fin de chaque bloc de données codé, où la génération des bits de contrôle de parité est créée par les données d'origine traversant une matrice creuse. Dans la norme 3GPP, LDPC prend en charge plusieurs tailles de matrices creuses où plus la taille de la matrice est large, plus le temps de traitement est long pour une même chaîne d'entrée en raison du nombre de bits de contrôle de parité à générer. Il existe deux matrices « Base Graph » (BG) qui ont des tailles de quarante-six fois soixante-huit pour BG1 et quarante-deux fois cinquante-deux pour BG2 pour le LDPC selon la norme 3GPP. Et l'expansion pour ces BG peut aller jusqu'à trois cent quatre-vingt-quatre, ce qui signifie que chaque nombre au BG se transformera en une matrice au niveau du bit avec une taille de facteur d'expansion multipliée par le facteur d'expansion. Quant au décodage, il utilise également la même matrice pour l'opération mais estime la chaîne de bits à travers le rapport log-vraisemblance (log-likelihood ratio, LLR). Le codage aussi bien que le décodage sont des opérations basées sur les blocs. Chaque bloc n'est pas lié aux autres, ce qui rend l'opération très répétitive et indépendante des données à travers les blocs. Étant donné que LDPC a toujours de grandes matrices de parité, l'opération matricielle prend du temps pour terminer un bloc. L'encodage et le décodage LDPC consomment à la fois du temps et des ressources. Il s'agit même des opérations les plus coûteuses en temps quand on teste tout le système en simulation. Le traitement matriciel basé sur des blocs et le long temps d'exécution en

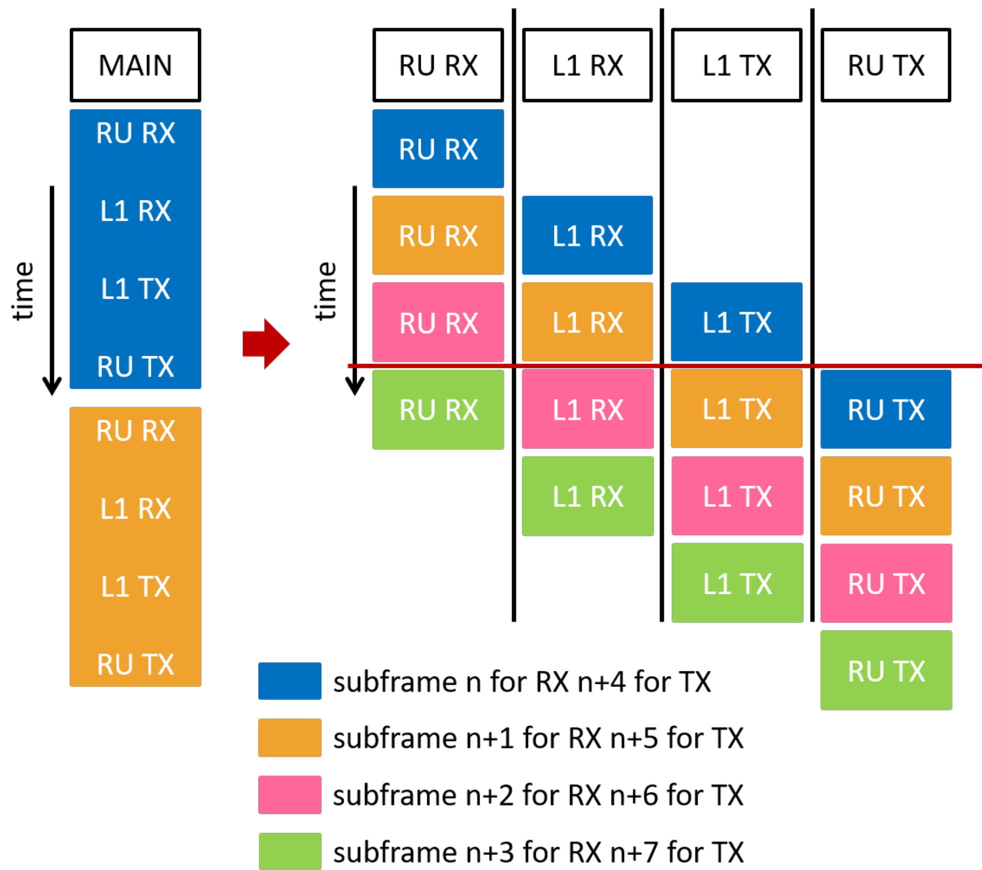

Figure 13 – Pipeline radio logicielle

font des candidats idéals pour l'accélération fonctionnelle.

Ce que fait le brouillage, c'est réduire la possibilité que des données proches soient contaminées en même temps, ce qui rend l'information plus difficile à décoder ou même impossible à inverser. Pour ce faire, le bit actuel est lié à un couple spécifique de bits qui ont déjà traversé le brouilleur auparavant. Après avoir passé la séquence de bits à travers une transformation de type aléatoire, on obtient la séquence de départ pour le brouillage. Puisque l'entrée se présente sous la forme d'une chaîne de bits et que chaque bit de sortie est affecté par le bit d'entrée précédent, il n'est pas approprié d'effectuer le brouillage en parallèle en raison de son exécution séquentielle native.

Quant à QAM, il fait correspondre un groupe de bits à un ensemble spécifique d'amplitudes et d'angles où l'amplitude et l'angle sont décidés par le type de QAM. Le niveau pour QAM peut aller par exemple du taux de modulation le plus bas, un à un, la modulation par déplacement de phase binaire (Binary phase-shift keying, BPSK), pour transporter plusieurs bits avec un ensemble d'amplitudes et d'angles, comme illustré à la figure 6. Une fois le niveau QAM décidé, les données de la chaîne seront alors regroupées puis transposées sur l'ensemble angle-amplitude. Cela signifie qu'il n'y a pas de relation entre les groupes, ce qui facilite le traitement parallèle du fonctionnement QAM. Cependant, ce que fait QAM, c'est simplement mapper des ensembles de bits sur des ensembles d'amplitudes et d'angles, ce qui ne prend pas beaucoup de temps.

Enfin, c'est OFDM. Il est utilisé avant d'envoyer un signal aux antennes pour résoudre les problèmes d'ISI et de « delay spread » comme mentionné précédemment. Dans la 5G, la transformée de Fourier rapide (FFT) et la transformée de Fourier discrète inverse (IDFT) pour le fonctionnement OFDM ont jusqu'à quatre mille quatre-vingt-seize points, ce qui signifie qu'il y aura une matrice de taille quatre mille quatre-vingt-seize fois quatre mille quatre-vingt-seize impliquée dans l'opération. Le traitement OFDM est également une opération matricielle avec une complexité en $O(n^2)$. Il existe des moyens de simplifier l'opération, mais même avec un meilleur algorithme, on obtient toujours une complexité en $O(nlogn)$. Comme il existe une taille maximale pour FFT et IDFT, la chaîne de données est séparée en groupes pour l'opération. Étant donné que la 5G utilise de multiples antennes, avoir toutes les données pour différentes antennes n'est pas efficace. La FFT et l'IDFT ont toutes deux un fonctionnement matriciel très complexe et le fonctionnement sur différentes antennes n'a aucune dépendance. Tout cela en font de bonnes candidates pour l'accélération.

Après avoir choisi le candidat pour l'accélération fonctionnelle, plusieurs méthodes existent pour réaliser l'accélération. La première consiste à avoir une accélération sur un CPU à plusieurs cœurs à l'aide de threads de travail. La seconde consiste à utiliser un matériel spécial pour effectuer le traitement (offloading). Et la troisième est d'avoir un jeu d'instructions spécial.

Pour l'accélération à l'aide de threads de travail, plusieurs threads de travail sont créés et utilisés par le thread d'origine qui est un thread dominant. Il contrôle les threads de travail qui traitent les données et les collecte à la fin du processus. C'est ainsi que le threading fonctionnel parallèle est utilisé. Ceux qui ont le fonctionnement de groupe, ce qui signifie qu'il n'y aura pas de dépendance au sein des groupes, pourraient alors être facilement séparés. Utiliser des threads de travail pour répartir la charge de travail pour

les processus les plus chronophages du pipeline est ce que nous voulons faire avec la radio logicielle temps réel. Comme mentionné précédemment, le codage et le décodage sont les goulots d'étranglement du traitement PHY. En analysant la façon dont les données sont gérées dans la fonction, il est clair que le codage et le décodage sont des opérations basées sur des blocs, ce qui signifie que l'on peut diviser quelques-uns des blocs en threads de travail pour obtenir du parallélisme. Le thread principal s'occupe des opérations communes avant l'opération principale, puis distribue une partie des données aux threads de travail qui ont des opérations identiques à exécuter en parallèle. Le temps d'exécution total sera le temps de fonctionnement commun plus le temps d'exécution du traitement des groupes de blocs qui est le temps d'exécution d'origine divisé par le nombre de threads de travail, plus enfin le temps d'attente des threads de travail terminant leur propre opération et signalant au thread principal que leur travail est terminé. En théorie, la meilleure performance que nous pouvons obtenir est le temps d'exécution des opérations communes et une part du temps d'exécution des processus parallèles.

Les processus DFT et IDFT sont tous deux des opérations basées sur des blocs et les données entre différentes antennes sont indépendantes. Cela rend le traitement séquentiel des différentes antennes inefficace. La parallélisation des traitements sur les antennes est l'une des façons de faire, en utilisant les threads correspondants pour effectuer les opérations IDFT et DFT par antenne, puis envoyer les données au fronthaul pour que l'antenne envoie le signal. En faisant cela, on passe du temps de traitement d'origine d'une antenne multiplié par le nombre d'antennes à peut-être juste un peu plus que le temps de traitement d'une antenne en raison de la surcharge due à la synchronisation. Le résultat de la combinaison de la structure du pipeline et des threads de travail est illustré à la figure 14.

L'utilisation d'un accélérateur externe, par exemple un FPGA ou un GPU, est la deuxième option. La plupart du temps, l'accélérateur externe a sa propre manière d'organiser les données, donc avant d'envoyer des données à un accélérateur externe, les données doivent être réorganisées sous forme de blocs. Ce n'est qu'alors que l'on pourra envoyer les données pour un traitement parallèle et puis collecter les données une fois l'opération dans l'accélérateur externe terminée. L'accélérateur externe doit alors renvoyer les données à l'étape du pipeline. Les données renvoyées peuvent nécessiter un autre réarrangement pour l'étape suivante. Il y a plus de données qui sont échangées lors de l'utilisation de l'accélérateur externe, ce qui signifie que nous devons nous concentrer sur l'opération la plus longue. Ce qui suit va parler de l'utilisation du GPU comme accélérateur externe. Le GPU fait naturellement de la programmation parallèle, ce qui lui donne un avantage pour accélérer un traitement dupliqué. Et cela fait du GPU un bon candidat pour le codage et le décodage. Mais comme mentionné précédemment, toutes les données doivent être réorganisées puis envoyées au GPU pour qu'il puisse les traiter. En particulier dans les stations de base des centres de données, les données transmises vers et depuis le GPU créent une longue latence. Cela signifie que l'utilisation de bus normaux pour le transfert de données n'est pas suffisante. Il y a besoin de bus à haut débit, supérieur à 1 Gbit/s, en radio logicielle pour un centre de données 5G. Tout cela signifie que la latence créée par le transfert des données devra également être prise en compte. Bien que le GPU ait un support natif pour la programmation parallèle, il a

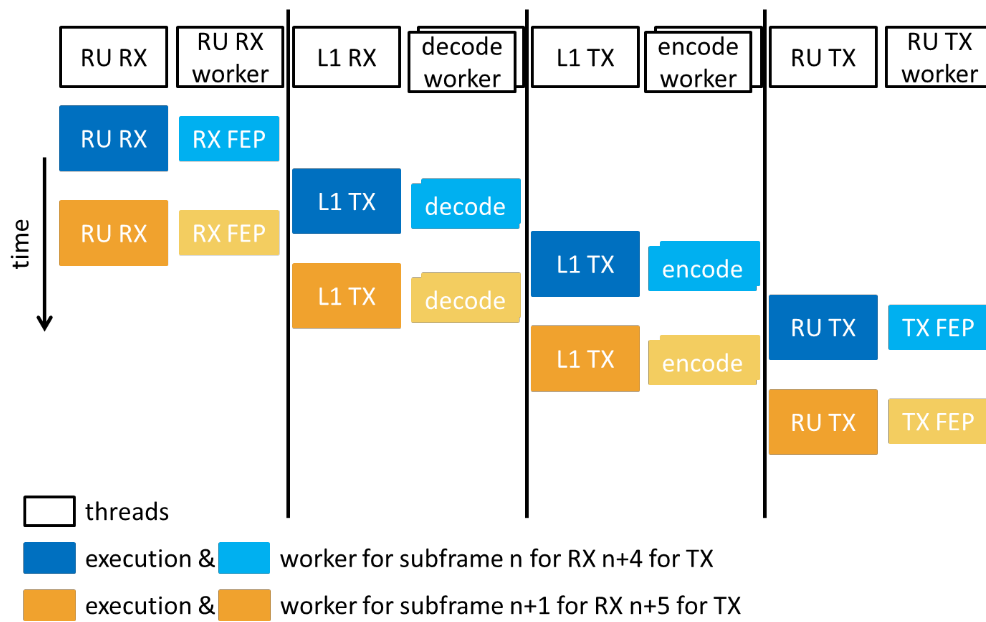

Figure 14 – Combiner les threads de travail et la structure du pipeline

une fréquence d'horloge plus lente, des cœurs plus simples, une mémoire partagée au sein d'un groupe de processeurs et plusieurs groupes de processeurs. Tous ces éléments rendent la programmation sur GPU plus difficile à concevoir et nécessitent plus d'efforts sur la conception parallèle.

La dernière technique d'accélération est d'utiliser un jeu d'instructions spécial. SIMD est l'abréviation de Single Instruction Multiple Data (Instruction Unique Données Multiples), ce qui montre déjà ses capacités de traitement parallèle natif de par son nom. Il s'agit d'un traitement parallèle au niveau de l'instruction. SIMD est utilisé pour traiter plusieurs variables à la fois. SIMD utilise l'idée d'avoir plusieurs opérations simples identiques opérant sur un grand vecteur avec une opération simple. Par exemple, l'opération d'addition de tous les éléments de « a » et « b », et de stockage du résultat dans « x » peut être transformée en un processus vectoriel « X = A + B ». C'est ainsi que SIMD fonctionne. SIMD prend en charge un groupe de données en y effectuant la même opération, par exemple ici une addition vectorielle. SIMD accélère en transformant une opération de boucle en une instruction opérant directement sur un vecteur. Cela raccourcit le temps d'exécution d'une boucle d'instructions au temps d'exécution d'une instruction SIMD.

Comme mentionné précédemment, il existe plusieurs endroits qui utilisent des blocs comme base de traitement et opèrent de manière répétée dans le traitement du signal. De nombreux traitements sont des opérations basées sur les bits. Le remplacement du code d'origine par un équivalent SIMD réduit considérablement le temps de fonctionnement pour le traitement de données de chaîne simples. Cela rend SIMD pratique pour accélérer considérablement le traitement. Partout où il y a une opération matricielle, il est possible

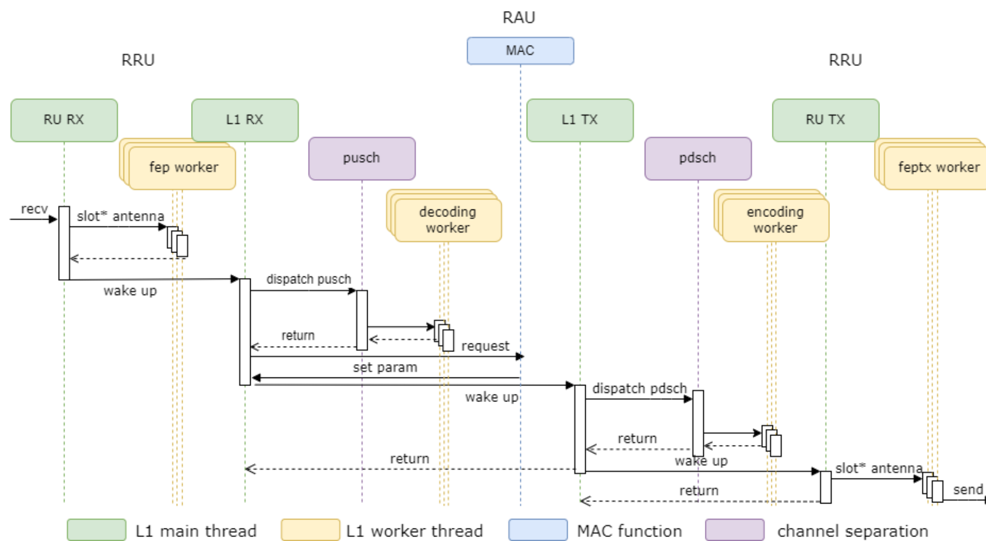

Figure 15 – La combinaison des threads de travail et de la structure du pipeline

de la transformer en plusieurs opérations vectorielles plutôt que d'utiliser une double boucle pour couvrir l'opération matricielle. En ce qui concerne la répétition d'opérations sur des données de chaîne, tant que l'opération elle-même n'entraîne pas de dépendance des données sur la chaîne de bits, il est également possible de dérouler l'opération en une opération vectorielle. Par exemple, LDPC est une opération matricielle qui peut être découpée en plusieurs processus vectoriels.

Pour SIMD, il y a plusieurs avantages et inconvénients. L'avantage le plus important que nous en tirons est de raccourcir considérablement le temps de traitement, ce qui compte beaucoup dans la radio logicielle temps réel. Pourtant, il y a encore des limites au SIMD. Il s'agit d'un jeu d'instructions spécial qui cible spécifiquement certains processeurs, ce qui signifie que la radio logicielle aura besoin de plusieurs jeux d'instructions SIMD correspondant aux différentes plates-formes prises en charge par le logiciel. Le jeu d'instructions SIMD ne prend en charge que les opérations simples, ce qui signifie que si l'opération est trop complexe à désassembler, elle ne peut pas utiliser SIMD pour s'exécuter. De plus, SIMD a des limites sur la longueur des données sur lesquelles il peut fonctionner. Cela signifie que si une donnée est plus longue que la taille des données SIMD, il faudra la séparer en plusieurs données pouvant tenir dans la taille des données SIMD pour pouvoir fonctionner. Le code SIMD est plus proche du code d'assemblage que du code C normal. Cela signifie qu'il est moins instinctif et plus difficile à comprendre. Il faut aussi transférer les données dans les registres spéciaux SIMD avant de pouvoir procéder. De plus, les données étant traitées par bloc, cela signifie que leur taille doit être multiple de la taille d'un bloc SIMD. Et comme il s'agit plus d'un code assembleur que d'un code C général, il est fortement dépendant de l'environnement ce qui rend difficile le changement de plate-forme.

Après avoir combiné toutes les fonctionnalités ci-dessus, on obtient une structure de radio logicielle comme illustré à la figure 15.

Acknowledgements

Contents

Abstract i
Acknowledgements xvii
Contents xix
List of Figures xx
List of Tables xxiii
Acronyms xxv
Notations 1

1 Introduction **1**
1.1 Software Defined Radio 6
1.2 5G Systems and Implementation Technologies 7
1.3 Motivation 10

2 Software Optimized Real-time Processing for 5G systems **11**
2.1 Related Work and State of the Art 11
2.1.1 Multi-threading for Parallelized Execution on Multi-Core Systems 12
2.1.2 Hardware Acceleration(FPGA/GPU) 13
2.1.3 SIMD/VLIW optimizations 14
2.2 Real-time SDR and Fronthaul Interface 15
2.3 Challenges in Implementing Real-time SDR 16
2.4 Considered Approach for Implementing Real-time SDR-base 5G systems . 16

3 Functional Decomposition and Pipelining for 5G radio processing **19**
3.1 5G Physical-Layer Procedures 20
3.2 Proposed Pipelining Methods 22
3.2.1 Pipeline Mechanism 23
3.2.2 Software Pipeline 24
3.2.3 Proposed Pipeline Structure 27

4 Acceleration Methods for 5G Functional Blocks **29**
4.1 Acceleration Target Choice 29
4.2 Splitting Method 34

4.3 SIMD optimizations 37

5 Implementations Using OpenAirInterface 39

5.1 Testing Methods 41

5.2 Performance Evaluation Methods 42

5.3 Results of Performance Evaluation on multi-core x86 platforms 45

6 Conclusion and Future Work 55

List of Figures

3 IMT-2020 ii
4 Inter Symbol Interference (ISI) iii
5 Inter Carrier Interference (ICI) iv
6 Quadrature Amplitude Modulation (QAM) diagram example iv
7 Flux de procédure PHY pour la radio logicielle vi
8 Réponse du canal dans différents modes vi
9 Procédure pour l'exécution séquentielle radio logicielle vii
10 Mappage des canaux - ShareTechNote viii
11 Un exemple de structure de pipeline à trois étages ix
12 Risque x
13 Pipeline radio logicielle xii
14 Combiner les threads de travail et la structure du pipeline xv
15 La combinaison des threads de travail et de la structure du pipeline . . . xvi

1.1 Internet seven layers from SystemZone 2
1.2 Inter Symbol Interference 3
1.3 Inter Symbol Interference 3
1.4 Three dimension for 5G from IMT-2020 4
1.5 SDR 5G network from IEEE [1] 5
1.6 OpenAirInterface Functional split 9
1.7 Flow for Physical Layer 9

2.1 OpenAirInterface Functional split 12
2.2 Hazard that might happen while using multi-thread 13
2.3 Four slot delay from received to send 15
2.4 Data-center environment with FPGA/GPU accelerator 17

3.1 Consecutive execution for receive and transmit set 20
3.2 Channel mapping diagram from ShareTechNote 21
3.3 Physical layer flow with the functional split 22
3.4 Simplest pipeline example 23
3.5 Hazard that might happen when using parallel programming 24
3.6 Example of One of The Mutex Used in SDR 25
3.7 Bit-wise mapping mask indicator 26
3.8 Threads communication with mutex lock introduced 27

3.9 Pipeline structure works in timing aspect 28

4.1 Resource grid for NR from ShareTechNote 30
4.2 Quadrature Amplitude Modulation (QAM) diagram example 33
4.3 OFDM effect on resource element for both time and frequency domain from 5G Technology World 34
4.4 Including worker threads to the procedure 35
4.5 Combing worker threads and pipline structure 36
4.6 The communication between threads when combining pipeline structure and worker threads 37

5.1 The role that Operation System (OS) play 40
5.2 Timing measurement on different threads 43
5.3 VCD tracking for function and variable 44
5.4 VCD tracking for function and variable 46
5.5 Two different pipeline split 47
5.6 Functions that is categorized in different acceleration group 47
5.7 The ability of pipeline structure to recovery from a sudden data rush . . . 49
5.8 The timing log of the recovery of the pipeline structure 49
5.9 Channel simulation result 51
5.10 Scenario that have multiple RRU in the pipeline structure 52
5.11 Timing analysis of Transmitting and Receiving every OFDM symbol . . . 52
5.12 An example of having more threads than cores 53
5.13 Front end process with single thread 53
5.14 Front end process with two thread 53
5.15 Execution resource distribution 54

List of Tables

1 Numérologie et espacement des sous-porteuses dans la norme 3GPP . . . v

1.1 Sub-carrier spacing and cyclic prefix under different numerology 4

5.1 N320 Test result on 60MHz bandwidth 50
5.2 N310 Test result on 60MHz bandwidth 50

Acronyms and Abbreviations

The acronyms and abbreviations used throughout the manuscript are specified in the following. They are presented here in their singular form, and their plural forms are constructed by adding and *s*, e.g. RRU (Remote Radio Unit) and RRUs (Remote Radio Units). The meaning of an acronym is also indicated once, the first time appearing in the text.

3GPP	Third Generation Partnership Project
5G	Fifth Generation
4G	Fourth Generation
ADC	Analog-to-digital converter
AI	Artificial Intelligence
API	Application Program Interface
AWGN	Additive White Gaussian Noise
BBU	BaseBand Unit
BG	Base Graph
BPSK	Binary phase-shift keying
BS	Base Station
CA	Carrier Aggregation
CAPEX	Capital Expenditure
CDD	Cyclic Delay Diversity
CDMA	Code-Division Multiple Access
CN	Core Network
CPU	Central Processing Unit
C-RAN	Cloud-Radio Access Network
CP	Cyclic Prefix
CPRI	Common Public Radio Interface
CSI	Channel State Information
CSIT	Channel State Information at the Transmitter
CU	Central Unit
CUDA	Compute Unified Device Architecture
DAC	Digital-to-Analog Converter
DAS	Distributed Antenna System
DFT	Discrete Fourier transform
DL	DownLink

DMRS	Demodulation Reference Signal
DSP	Digital Signal Processor
DU	Distributed Unit
ECP	Extended Cyclic Prefix
eMBB	Enhanced Mobile Broadband
EPC	Enhanced Packet Core
FC	Fast Calibration
FDD	Frequency-Division Duplex
FPGA	Field Programmable Gate Array
FTT	Fast Fourier Transform
gNB	gNodeB
GPP	General-Purpose Processor
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HPA	High Power Amplifier
HSS	Home Subscriber Server
ICI	Inter-Carrier Interference
IDFT	Inverse Discrete Fourier transform
IFFT	Inverse Fast Fourier Transform
IoT	Internet of Things
IRIG	Inter Range Instrumentation Group
ISI	Inter-Symbol Interference
L1	Layer1
LDS	Laser Direct Structuring
LDPC	Low-Density Parity-Check
LTE	Long Term Evolution
MAC	Media Access Control layer
MCS	Modulation and Coding Scheme
MIMO	Multiple-Input Multiple-Output
ML	Machine Learning
MME	Mobility Management Entity
MMSE	Minimum Mean Square Error
mMTC	Massive Machine Type Communications
mmWave	Millimeter Wave
MRT	Maximum Ratio Transmission
MSE	Mean Square Error
MU	Multi-User
NLOS	Non Line Of Sight
NR	New Radio
OAI	OpenAirInterface
OCXO	Oven-Controlled Crystal Oscillator
OFDM	Orthogonal Frequency Division Multiplexing
OFDMA	Orthogonal Frequency Division Multiple Access
OS	Operation System

OSI	Open System Interconnection
SIMD	Single Instruction Multiple Data
PA	Power Amplifier
PBCH	Physical Broadcast Channel
PCB	Printed Circuit Board
PCIe	Peripheral Component Interconnect Express
PDCCH	Physical Downlink Control Channel
PDCP	Packet Data Convergence Protocol
PDSCH	Physical Downlink Shared Channel
PHY	Physical Layer
PRB	Physical Resource Block
PSS	Primary Synchronization Sequence
PTP	Precision Time Protocol
PUCCH	Uplink Shared Channel
PUSCH	Uplink Control Channel
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase-Shift Keying
RAN	Radio Access Network
RAU	Radio Aggregation Unit
RB	Resource Block
RCC	Radio Cloud Center
RE	Resource Element
RF	Radio Frequency
RLC	Radio Link Control
RRC	Radio Resource Control
RRU	Remote Radio Unit
RS	Reference Signal
RU	Radio Unit
Rx	Receiver
SGw	Serving Gateway
SINR	Signal to Interference and Noise Ratio
SM	Spatial Modulation
SNR	Signal to Noise Ratio
SRS	Sounding Reference Signal
SSF	Special SubFrame
SyncE	Synchronous Ethernet
TDD	Time Division Duplex
TM	Transmission Mode
Tx	Transmitter
UE	User Equipment
UHD	USRP Hardware Driver
UL	UpLink
URLLC	Ultra-reliable and Low Latency Communications
USRP	Universal Software Radio Peripheral

UTC	Coordinated Universal Time
VCD	Value Change Dump
VCO	Voltage-Controlled Oscillator
VCTCXO	Voltage-Controlled Temperature Compensated Crystal Oscillators
VLIW	Very Long Instruction Word
ZF	Zero Forcing
ZFBF	Zero-Forcing Beamforming

Chapter 1

Introduction

The implementation of a real-time Software Defined Radio (SDR) for 5G is the topic of the thesis. Before learning more about real-time 5G SDR, let's discuss some background information. First Let's discuss the definition of the internet. As depicted in figure 1.1, there are seven separate layers in the Open System Interconnection (OSI) paradigm of communication. The Physical Layer (PHY), Data Link Layer, Network Layer, Transport Layer, Session Layer, Presentation Layer, and Application Layer are the layers in order from lowest to highest. Physical nodes are used to send and receive data throughout the network, and the physical layer is in charge of handling raw bitwise data strings. The Media Access Control (MAC) and Logical Link Control (LLC) are combined to form the data link layer, which controls whether connections between physical nodes are established or removed. The PHY was controlled by MAC, but LLC will make the corrections that the PHY missed. The thesis will make reference of these two. The Application Layer, the uppermost level, is what we are most familiar with. As the sole result most users are familiar with, it is in charge of delivering and receiving data directly from the users and showing the result to them.

An SDR is a radio communication system that primarily uses the bottom layer, as indicated by the name. SDR supports a variety of data transmission methods, including wireless, optical fiber, and any type of cable. Out of all of these, the SDR will focus on wireless communication for this thesis. The unique aspect of wireless communication is that, as implied by the name, it transmits data across the air without the use of wires or other conductors. The advantage of this is that there are less physical limits because no conductors are required for true connection, but the disadvantage is that there may be greater interference during the transmission. We must first discuss Orthogonal Frequency-Division Multiplexing (OFDM), one of the best communication architectures, before we can discuss wireless communication. It carries the data on a number of carriers in the frequency domain that are orthogonal to one another. Therefore, except for the case where the data is optimal, there will be cancellation everywhere else. However, in reality, inter-carrier interference (ICI) and inter-symbol interference (ISI) still exist. ISI and ICI happen while having a delay for the arrived package which means that there will be some data missing for the expected signal and being overlapped by the signal attached to it. By include a guardian period with known data, such as the zero padding illustrated

Figure 1.1 – Internet seven layers from SystemZone

in figure 1.2 could be resolved. However, ICI continues to exist because there is still a signal component that is lost during the process. While adding known data padding solves the ISI problem, it doesn't solve the ICI. Since the time shift in the time domain is only an offset in the frequency domain, as demonstrated in Figure 1.3, inducing Cyclic Prefix (CP) rather than known data padding produces the desired effect for ICI.

As previously said, SDR focuses on the lower layer more. While the Physical Layer will be the main focus of this thesis, the SDR is built for wireless communication for 5G as described in the 3gpp standard. Timing will be one of the biggest obstacles real-time SDR would have to overcome. Slot time is the most fundamental PHY unit, and table 1.1 in the 3GPP standard for 5G illustrates the link between subcarrier spacing and numerology. According to space-time transmission between slot time and subcarrier spacing, a rough idea of using wider subcarrier spacing will induce a shorter slot time. The following equations show the relationship between subcarrier spacing and slot time where slot time is the basic time unit in PHY. $T_c = 1/(\Delta f_{max} * N_f)$ were $\Delta f_{max} = 480 * 10^3$HZ and $N_f = 4092$, The constant $k = T_s/T_c = 64$ where $T_s = 1/(\Delta f_{ref} * N_{f,ref})$,$\Delta f_{ref} = 15 * 10^3$HZ and $N_{f,,ref} = 2048 T_f = (\Delta f_{max} N_f/100) * T_c = 10ms$. These equations demonstrate that although the frame time is always 10 ms, there will be more slots per frame as the numerology increases, resulting in a shorter slot time when utilizing higher numerology. The slot time in 5G is typically less than what is required in LTE because numerology zero has the same setup as LTE and has a slot time of 1ms.

Let's talk a little bit about the Fifth Generation of Cellular Mobile Communications (5G) New Radio (NR) network now that you know a little bit about the real-time SDR. As seen in figure 1.4, 5G NR provides a variety of features, such as Enhanced Mobile Broadband (eMBB), Ultra-reliable and Low Latency Communications (URLLC), and Massive Machine Type Communications (mMTC). Every component has a central

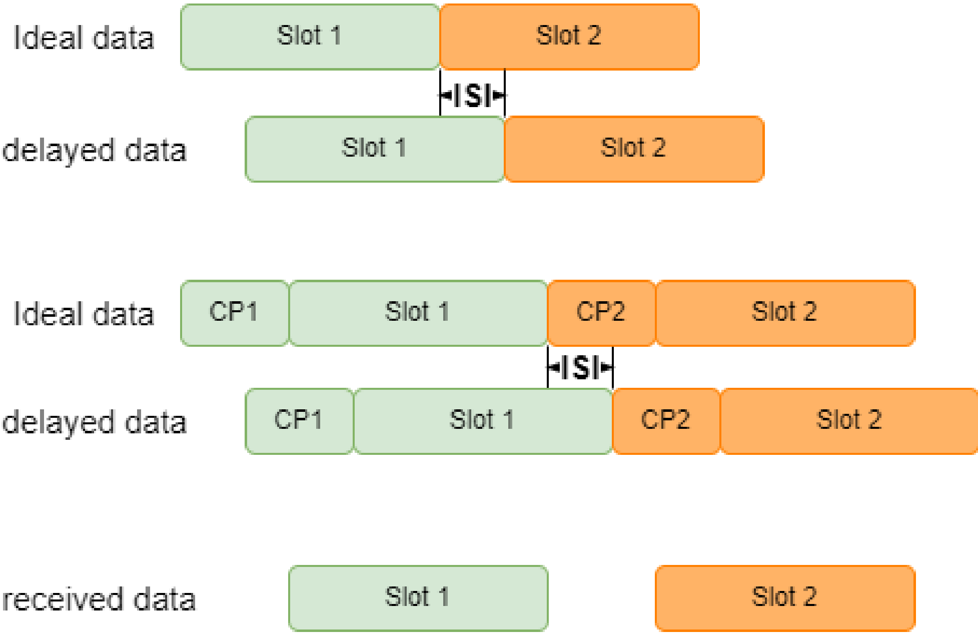

Figure 1.2 – Inter Symbol Interference

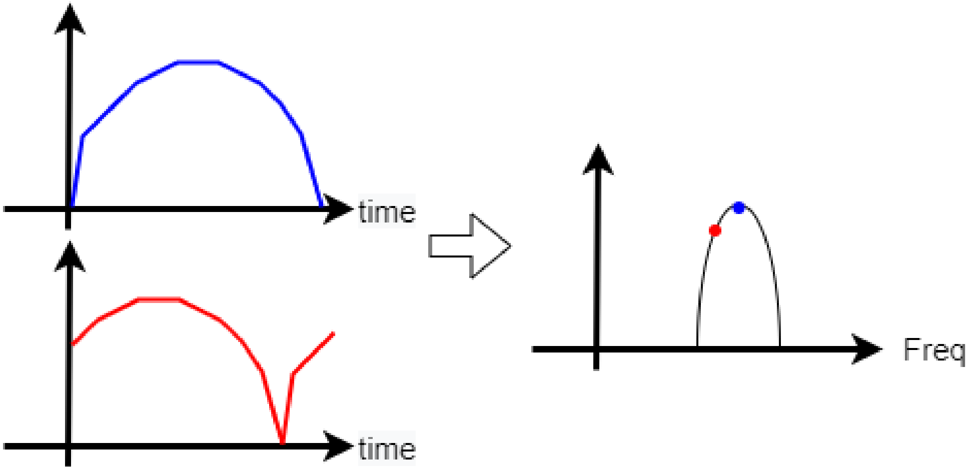

Figure 1.3 – Inter Symbol Interference

μ	$\Delta f = 2^{\mu} * 15[kHz]$	Cyclic prefix
0	15	Normal
1	30	Normal
2	60	Normal, Extended
3	120	Normal
4	240	Normal

Table 1.1 – Sub-carrier spacing and cyclic prefix under different numerology

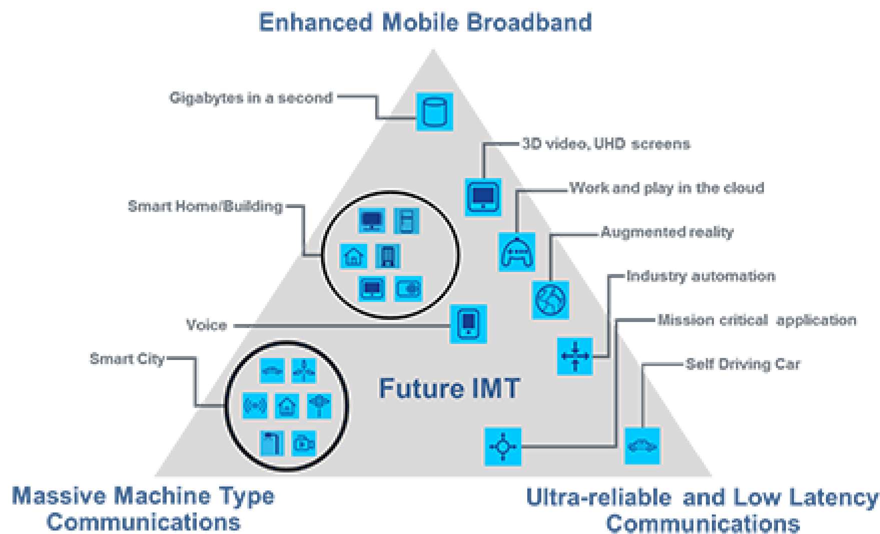

Figure 1.4 – Three dimension for 5G from IMT-2020

focus. eMBB is concentrating on high data rates over a wide band, whereas URLLC is concentrating more on precision and speed. With regard to mMTC, it focuses more on simultaneously supplying a sizable number of devices across a broad range.

According to standard 38.801, the use case for 5G NR is a centralized base station with multiple nodes feeding various antennae. Since the Central Unit (CU) may support multiple Distributed Units (DU) with a distance and several localized nodes, as shown in figure 1.5, it is anticipated that 5G NR would have a centralized deployment. This will allow the base station to have larger coverage. Additionally, DU is connected locally or remotely by multiple radio units (RU) or remote radio units (RRU). Particularly the Physical Layer (PHY), those processes in DU and RU have a larger demand for real-time property. High throughput and minimal latency are needed from a 5G base station, and one of the obstacles will rely on how well the DU can perform.

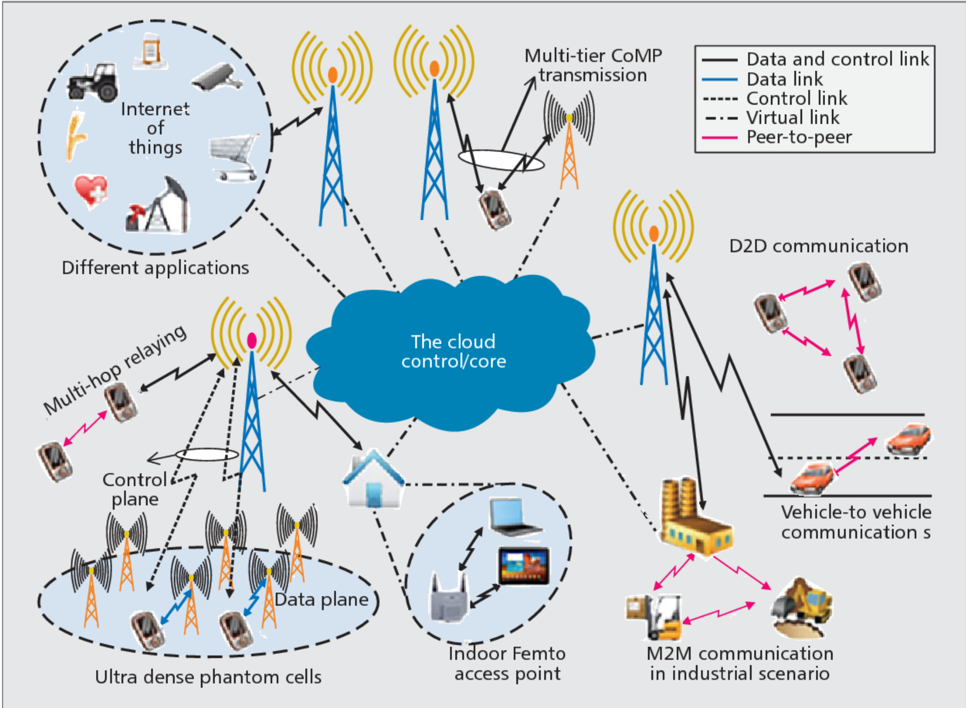

Figure 1. Schematic digram for 5G cellular network. The cloud is the core engine of the network which

Figure 1.5 – SDR 5G network from IEEE [1]

1.1 Software Defined Radio

The thesis focuses on developing a real-time Software Defined Radio (SDR) for 5G using citations from the following works [2] [3] [4] [5] [6] [7]. The thesis is about implementing a real-time Software Defined Radio (SDR). SDR does not have a single standard, but in this case it is a combination of software programs that connects with the hardware frontend. SDR, as its name suggests, is a program that runs on a computer that once housed a radio station. It stands out because it is a software application that performs tasks that were previously handled by hardware devices while having capabilities for software. One of the factors contributing to SDR's rising popularity is that a chip's extended development period allows for minimal room for error. To be able to create a semiconductor that functions, it must first design with a low-level language like Verilog for the behavior design which is not intuitive. The size of each gate, how it is implemented, and where it is positioned all matter since the size of the chip directly affects its price, so the program must be converted into visual form. Additionally, the connection path between each gate on the chip needs to be carefully constructed because a long link could result in problems like path loss, which is a power decrease when an electric wave travels a long distance, and data hazards which is data that is delayed. Additionally, this is only a quick look at the chip design at the graphics stage. After going through all of these steps, it needs to be taken to the manufacturing facility for production where the graphic is actually etched onto the chip. It must go through four phases in order to have one layer for the chip: coating, etching the coat to reveal the silicon, etching the silicon, and washing out the residue. This demonstrates that producing a chip requires time as well as precision. We can infer that making a chip is expensive and time-consuming from all the details given above. Hardware verification still requires more work than software, even without a chip. Here's an illustration of functional verification using an FPGA board. In addition to requiring the use of a low-level programming language, the FPGA's compilation time is significantly higher than that of a standard CPU because it must modify the board's gateway.

When SDR uses software programming, as in the cases of [8] [9], it offers the advantage of a quick development cycle and an easy-to-add new feature. However, SDR must still overcome some challenges, starting with the timing problem. The hardware chip went through all of these difficulties in order to deliver the signal as quickly as it possibly could, whereas general-purpose software programs can't match the performance, which was not even possible back in the day due to the slow clock rate and other factors. The good news for the SDR is that, despite the fact that the most recent software was still unable to match the performance of the most recent hardware, it was now capable of doing the SDR procedure within the time frame demanded by 3gpp. Modern technology has made it possible for the SDR to be served by a generic CPU. Despite this drawback, it has the unmistakable advantage of having a quick development cycle, which is apparent whenever a new feature needs to be introduced to the SDR. The SDR can easily handle the additional features if we simply add the new functions to the code and recompile it. It is not only easy to add new features, but it can also support multiple configurations [9] within one version of the SDR. Since the software program can be configured in several

ways, each configuration only needs to predefine a small number of variables. Because of SDR's development flexibility and understanding that it will still meet the timing requirement, more and more people are becoming interested in it.

As previously indicated, the SDR's strong adaptiveness speeds up development whenever new features are added to the standard. It is now necessary for the station to offer numerous functions at once because modern communication is more sophisticated and diversified than it used to be, which makes creating hardware even less appealing. SDR is also known as reconfigurable radio, which indicates from the name that it is made to be able to handle a variety of communication. Since general-purpose computers are becoming more powerful, hardware is no longer the only way to implement radio. SDR development will need to be flexible in terms of frequency, bandwidth, modulation, data rate, and other factors, so the lifespan of the device may be extended.

Although practically all of the hardware components of the SDR radio station have been replaced by software, the antenna is the one component that software will never be able to take the place of. Since the data stream must travel through the air with the analog signal for wireless communication, there must always be a digital-to-analog or analog-to-digital converter (DAC/ADC) by itself, in addition to the antennas. The communication between the RU and the rest of the SDR is becoming increasingly complicated and requires discussion since 5G must be a large Multiple-Input Multiple-Output (MIMO) system.

Combining all of the above will make it difficult to create an SDR that can handle the complex set necessary to meet the 5G specification alone with the MIMO usages while still being able to support the massive data flow within the timing requirement. This thesis will focus on this challenge.

1.2 5G Systems and Implementation Technologies

The three dimensions that 5G hopes to achieve are eMBB, URLLC, and mMTC, as was previously indicated. All three of the 5G implementation's dimensions, as depicted in 1.4, are covered in [10] [11]respectively. eMBB is the most understandable. By enabling a greater bandwidth and more PHY layers than the previous generation LTE, it is seen as an improved version of the LTE. This means that 5G might enable better data rates and more data flow. For instance, the implementation of virtual reality (VR) may have greater visual quality, live streaming may be more fluid and have higher resolution, and for activities requiring large amounts of data transfer. In addition to eMBB, there are two further new features over previous-generation LTE: URLLC and mMTC. Let's start by discussing URLLC. The short time unit feature of URLLC, which also translates to short slot time, is obvious from the name. In addition, there is hardly any chance that anything will go wrong. For precise implementation of tasks requiring quick response times, such as emergency medical treatment and, in particular, the auto drive system for vehicles as detailed in [12], URLLC is employed. The vehicle control instruction is rather straightforward, but every signal that the system picks up is important for the decision-making process, and the moving vehicle's quick reaction time makes it an ideal choice for using URLLC. Regarding mMTC, it is utilized to support a sizable number of

devices simultaneously, as its name suggests. The challenge is being able to handle so many devices at once, even while each one that is linked to the radio doesn't need a lot of resources. The idea of the Internet of Things (IoT), which connects numerous devices with centralized control, meets the definition of mMTC. One example is the healthcare system as stated in [13].

In comparison to LET, 5G requires more from all data strings supplied, execution time unit, users, data rate, capacity, signaling overhead, energy consumption, end-to-end latency, dependability, and other factors. The figure 1.5 [14] [15] shows that 5G has the structure of several wild distributed RU with a relatively centralized computation unit. The layout of which functions should exist in Cartelized Unit (CU) and which functions should exist in Distributed Unit (DU) becomes more crucial since the implementation of such a structure with the necessity of 5G entails a short time interval with large data. Due to the combination of DU and MIMO, a lot of data transmission time may be wasted, or the data could be sent with a minor modification that could be readily accomplished by making a small adjustment from the original data. Multiple strategies for splitting functions between CU and DU in OAI are shown in Figure 1.6. One of the most significant constraints, the Transmission Time Interval (TTI), was also described as the fundamental time unit in the chapter before, in addition to the latency caused by data transfer. Less latency and a shorter TTI should be achievable with the 5G SDR, which will result in faster PHY processing. To achieve the latency requirement, 5G PHY processing times should be quick. It should also have sufficient data processing speed to support MIMO. PHY needs to be adaptable in order to support various usage requirements in 5G.

Let's examine the data flow in PHY in more detail since it is crucial to have the split inside the data flow. The brief PHY flow is depicted in figure 1.7, which doesn't much change from that in LTE other than the use of a new strategy and algorithm. For instance, Low-Density Parity-Check (LDPC) encoding is used for coding and decoding in 5G instead of turbo encoding. Even though the fundamental concept hasn't changed much, there have been a lot of tiny adjustments made, such as the coding rate due to differing Base Graph (BG) widths used by LDPC, channel response timing, and most crucially, the TTI as previously indicated. Since there could be more than one layer in 5G, the amount of data travelling through the PHY is also variable. Additionally, MIMO is a native feature of the 5G framework, whereas Single-Input Single-Output (SISO) is only permitted on 4G. This increases the processing time for antennas by a few times compared to before. When multi-layer and MIMO are combined, layer mapping and precoding, which in 4G were merely utilized for data transfer, will need to be used as connecting elements.

A higher Fast Fourier Transform (FFT) size is employed in 5G in addition to a shorter TTI, different coding rates, and additional antennas and layers. This indicates that the coding process utilizes a larger matrix. The 3gpp standard states that the FFT size for 5G increases from 1024 to 4096 depending on the bandwidth and numerology since a larger numerology results in a shorter slot duration and a larger subcarrier spacing.

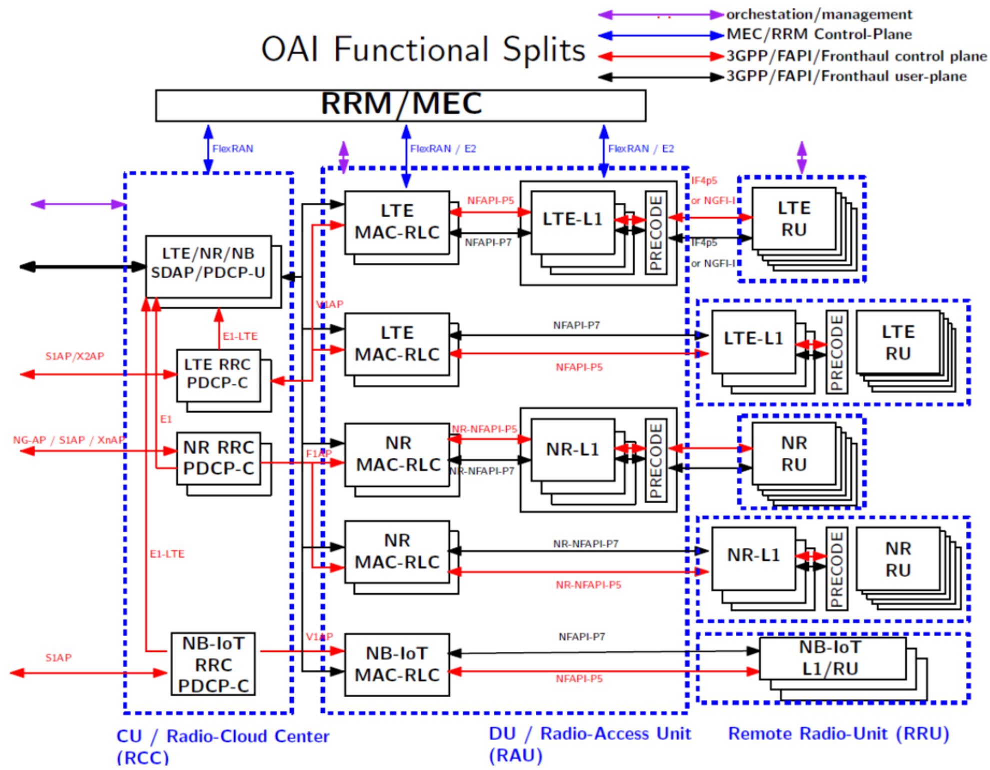

Figure 1.6 – OpenAirInterface Functional split

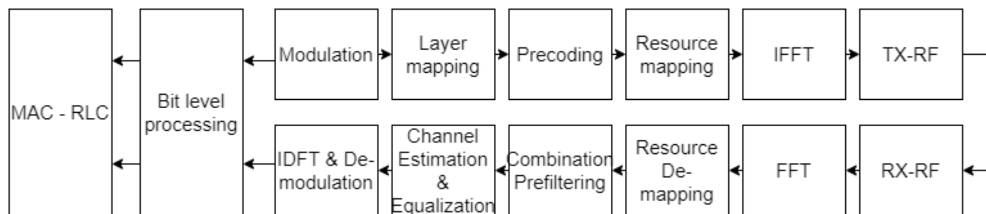

Figure 1.7 – Flow for Physical Layer

1.3 Motivation

From the summary above, it is clear that there would be a few challenges when constructing a 5G system with SDR. The two factors that matter the most out of all are having a lot of data to analyze and a fast PHY since MIMO uses numerology to apply multi-layer data to a broader bandwidth. The variety of 5G makes it increasingly difficult and impractical to create a specific chip. So it has become more appealing to create an SDR capable of supporting 5G. The SDR can begin testing with the 4G requirements before moving on to the 5G ones because 5G requires more timing and resources than 4G does.

Due to the lengthy processing time with limited data, approaching the behavior without acceleration is initially still manageable with 4G. The timing issue in PHY is then apparent even when just altering the configuration for the numerology in 4G and modifying the coding and decoding scheme are done. The original SDR architecture cannot handle the 5G operation by just porting the 3gpp standard. The question of how to make SDR work with the 5G time requirement becomes one that merits discussion.

PHY performance should be approached from a few different aspects, including speeding the process itself and performing structural rearrangement by overlapping the operations. Even though it sounds straightforward, multiple users processing data at once, handling data before it is available, and other similar problems might result from overlapping processes. When it comes to speeding up the operation itself, creating a faster algorithm doesn't appear to be a viable option. Instead, job offloading or having special instructions for quick execution come into play.

Chapter 2

Software Optimized Real-time Processing for 5G systems

One of the open-source real-time and simulation SDR base stations, known as OpenAir-Interface5g (OAI), which was initially built on LTE and is currently being integrated and developed in 5G NR that adheres to the 3gpp protocol. The SDR in this thesis will be implemented using OAI. The advancement of the present chip development makes real-time SDR viable in comparison to the old hardware-based base station. For real-time SDR, it is most important to have software processing time to be able to match up the timing required for the front end for it to not only be able to do the verification for the procedure but actually be a base station [16]. According to [17], matching the front end time means that in addition to all the software signal processing, it must be able to handle the input and output throughput from the front end.

When building the base station using software, one of the most crucial difficulties is the real-time programming paradigm. The corresponding figure 1.6 illustrates how splits for 5G NR are implemented in OAI.

We can deduce from the aforementioned figure that all signals and data are transmitted one at a time, leading to the instinctive sequential design. A real-time program does not need a lot of idle resources or pending time, though, which is what consecutive programming will do. Therefore, the main focus of this issue is on how to shorten the waiting time and maximize the use of the resource.

2.1 Related Work and State of the Art

in order to keep up with real-time. There are a few ways to speed things up, including hardware acceleration, threads that use cores, and programming language configuration. According to the source [18], SDR can be based on a variety of platforms, including General-Purpose Processors (GPPs) and Field Programmable Gate Arrays (FPGAs). A lower-level programming language is utilized to enable real-time programming for SDR. The hardware language Verilog must be used to program FPGAs, whereas C is used to program GPPs. Hardware language is more focused on programming with fine-

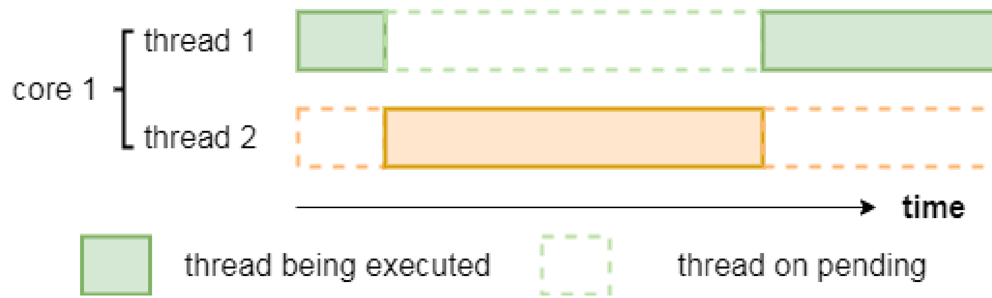

Figure 2.1 – OpenAirInterface Functional split

grained logic gates and wire connections, whereas C language is more focused on logical programming with easier to maintain data structures, command libraries, and cross-platform portability.In order to design the SDR, we therefore selected the General-Purpose Processor with C language for development and programmability.

2.1.1 Multi-threading for Parallelized Execution on Multi-Core Systems

First, let's discuss cores and threads. The ones on the chips that are physically present are called cores. The ability of each core to function independently makes cores in parallel functional. Each physical thread has two cores. They are also utilized to enhance compute performance by parallelization, which is the simultaneous execution of numerous tasks. Threads cannot do many tasks simultaneously, unlike cores. However, while tasks are being handled by a core, if tasks in thread one of the core are pending, such as waiting for data access in memory, the core can then transition to duties in thread two to make the most of the pending time that thread one creates, as seen in figure 2.1. The overall thread count is equal to the number of cores multiplied by the maximum number of threads that each Intel core can support under its current architecture, which is two.

However, when discussing multi-threading programming, software risk is a given. A process will be more likely to experience hazards if it manages several threads. Hazards include consulting data before handling it or cross-changing data in several threads, which can lead to inaccurate answers. The figure 2.2 displays a few instances of the risk occurring. Reading and writing simultaneously on the same register or performing more than one act of writing simultaneously on the same register may produce surprising results since the data may be overwritten at the wrong time. In addition, any work that follows branch merging can only be started until all branches ahead have been completed. The adage "the more threads, the faster it is" won't always hold true because there will be communication overhead between threads. Code complexity and readability will both be drawbacks of multi-threaded programming.

The aforementioned factors need taking extra care while employing several threads for the same activity. Data being processed for multi-thread parallelization should be carefully organized because only unrelated data can be executed on different threads at the same time safely without running into issues. In addition to a sign or a check to

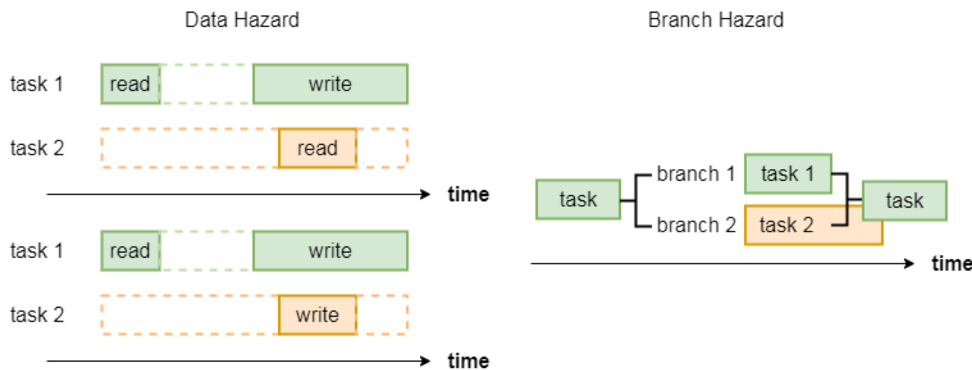

Figure 2.2 – Hazard that might happen while using multi-thread

signify the start of parallelized execution, there should also be a guard for common usage data, such as signal sending to activate parallelized execution.

[19] is cited as saying that performance increases rapidly as core count increases. In order to create an SDR, [19] combines multiple Digital Signal Processors (DSPs). This eliminates the need for the engineer to modify the algorithm in order to fit it into the resource of a single DSP and achieve real-time performance, which occasionally may entail sacrificing the algorithm's accuracy. The question of how the algorithm might be divided has taken precedence. However, because of the overhead between DSPs, the relationship between performance and core count has not grown linearly. It nonetheless demonstrates how having more than one processor can boost performance, despite the fact that it employs DSPs rather than the General Purpose Processor's (GPP) cores.

Real-time reaction, environmental adjustment, and decoding all have various duty cycles and levels of parallelism (data level, instruction level, and task level), as noted in [20]. Despite adding complexity to the SDR by nature, this makes multi-core work load sorting and splitting acceptable. Not to mention the simultaneous existence of numerous modes in the same SDR. Using several cores for the SDR process maximizes resource utilization, which can lower energy usage and enhance timeliness.

2.1.2 Hardware Acceleration(FPGA/GPU)

We can infer that multi-task processing is required for present SDR from the part above. However, communication and synchronization issues arise with parallel execution. To prevent stalling, the SDR's organizational structure and resource consumption should be well-planned. Adding the attached hardware accelerators is an alternative to utilizing the current cores, citing [21].

Hardware with image processing and graphics capabilities is known as a graphics processing unit (GPU). The handling of an enormous quantity of data in the same way is a natural feature of the graphical process. And as a result, when stream data processing is designed properly, GPUs are effective at it. Due to this, GPU is a strong contender when discussing data-level parallelism. For instance, NVIDIA's Compute Unified Device Architecture (CUDA) GPU always has a structure with numerous groups of cores that is

less powerful than the general processor, and each core has its own unique memory with shared memory inside the group. Providing a GPU with a single piece of data in a single motion, in my experience, takes almost as long as providing a GPU with several inputs in a single move, as in the case of an array process. The configuration of data input and output for GPU will then be the primary distinction.

[22] A programmable hardware accelerator can also be a Field Programmable Gate Array (FPGA) that has been Hardware Description Language (HDL) programmed. Having a dedicated chip for a specific use was a possibility when considering unit acceleration. However, manufacturing a chip is time-consuming and expensive; using programmable hardware, such as FPGA, instead, is more cost-effective. By substituting an FPGA for the existing component, the unit that has to be accelerated can be used exclusively. The chosen component is then redesigned, wired, and connected to the remainder of the process inside the FPGA. This results in the processing of data inside the unit being more conventionally analog rather than digital, which will speed up the process.

A case of combining FPGA with conventional CPUs is the article [23]. It offers multiple modes where the FPGA controls specific parts of the overall operation. The processing time for the FPGA is faster than that of the general processor handling the same process in a different mode, despite the fact that it differs from using FPGA just for acceleration.

2.1.3 SIMD/VLIW optimizations

It is assumed that repetitive execution will be held on various data in SDR because communication protocols are about processing the stream of data in a consistent manner. Single Instruction Multiple Data (SIMD) is now a possible acceleration method as a result. By nature of its name, SIMD makes it clear that it handles many data sets simultaneously, which is what communication processing requires for. SIMD is parallelism at the instruction level. A number of data should be combined into one large piece of data and then handled by special instructions based on the language and data size, such as AVX256, in order to be able to analyze numerous data at once.

The article [24] claims that vector data is used for SIMD operation. Due to the fact that it is a specific process, alignment was needed for the data being served, which usually involves data movement. The barrier for SIMD instructions is now data organization rather than execution time. The issue of complex vector computation is then resolved with the advent of Very Long Instruction Words (VLIW). Compared to SIMD, VLIW takes a different approach to acceleration. It benefits from the fact that all data goes through the same procedure, but the data has dependencies that prevent them from splitting in parallel as usual. Process is divided into smaller chunks and has a pipeline structure for it rather than employing typical consecutive execution. Whenever greater accuracy is required The concurrent execution that the SIMD pipeline couldn't manage will subsequently be handled by VLIM.

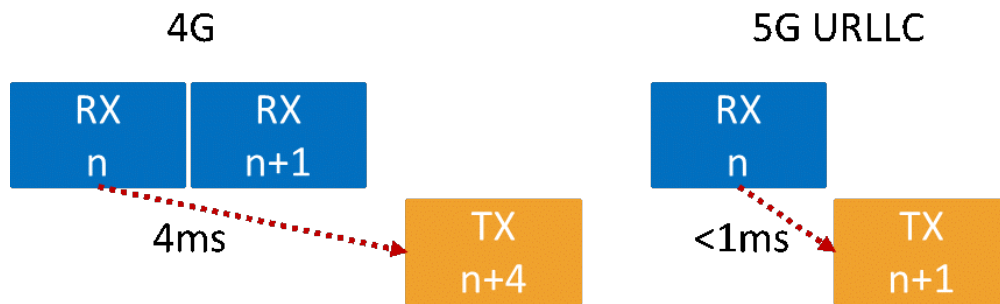

Figure 2.3 – Four slot delay from received to send

2.2 Real-time SDR and Fronthaul Interface

SDRs handle digital data, but the signal that is actually delivered is an analog waveform. To convert analog signals to digital data and the other way around, there must be a data transformation process between digital and analog that includes sampling. Antennas are furthermore required for wireless transmission. For the SDR to function fully, all of these indicate that at least one piece of hardware must be connected. We refer to the period prior to the Base Band Unit (BBU) as fronthaul or Remote Radio Unit (RRU). Fronthaul should operate in real-time since the SDR does. This requires not only that fronthaul processing be completed on time, but also that data transfer from RRU to BBU be accomplished in real-time.

Transmission Time Interval (TTI) could be substantially lower than in LTE because of the increase in subcarrier spacing brought on by the numerologies, according to the 5G standard (38.211). According to the specification, real-time SDR must have a consistent input and output of slot data per TTI, which in 5G is lower than 500ms. Furthermore, 5G NR has stricter requirements for low latency, particularly for URLLC, as is illustrated in figure 2.3. The amount of data traffic in 5G NR has also grown, particularly for Enhanced Mobile Broadband (eMBB), making it crucial to consider scheduling for data delivery even to the offloading device.

Because MIMO is necessary for 5G, the loading for real-time fronthaul is larger. When it used to send just one string of data every TTI, it now needs to send a few times more data while still getting the job done in a single TTI. There are primarily two options: either all Radio Units (RU) or both the lowest layer of PHY and RU are present in the fronthaul structure. In the first situation, each antenna's data must be sent via BBU within a single TTI. For the second, each layer's string data will need to be transmitted using BBU.

The Universal Software Radio Peripheral (USRP) family is utilized as the fronthaul of the SDR in the article [25]. The feature of USRP is also discussed in the Ettus Research article [26]. Gigabit Ethernet or USB 2.0 ports on the processor are available for connecting USRP. USB 2.0 is no longer effective enough to support the real-time functionality of 5G; only high-velocity Ethernet, like 10 Gigabit Ethernet, can keep up

with the speed for real-time SDR. USRP hardware includes an RF front end, FPGA, ADC, and DAC. Furthermore, Ettus created the USRP Hardware Driver (UHD), an application program interface (API) to achieve real-time performance. 5G uses a variety of configurations for its bandwidth, bands, frequency, antennas, and other components, so the fronthaul must also be programmable.

2.3 Challenges in Implementing Real-time SDR

As previously noted, having the simplest concurrent dataflow with no acceleration required and just software acting as a sandbox in simulation mode can be achieved to have an SDR for development and logical verification. However, achieving real-time is the most crucial aspect of real-time SDR. The TTI, which in 5G is less than 500us, must be the basis for all process unit times in the physical layer. Things become more challenging as a result. It is necessary to reconstruct the PHY structure, which cannot be sequential. The shorter the processing unit, the better. Before the reaction time, the corresponding data needs to be ready. Not only these, but the data rate, capacity, signaling overhead, end-to-end latency, energy consumption, and number of devices have all been improved in the 5G system. Real-time SDR implementation must satisfy all of the aforementioned requirements, which makes calculating and allocating computing resources for the general-purpose computer the primary challenge. Finding and understanding the timing critical path is crucial since Real-time is the implementation constraint. However, there might be several executions going on at once, and speeding up the existing critical path might not be sufficient. It becomes crucial to consider how the process will run and how much work it would require.

In conclusion, there are going to be a few points that need to be taken care of in the real-time SDR.

- Structure of the SDR – Rearranging the structure to parallel programming. Observing critical path.
- Unit process time – Shorten up each unit time using different levels of parallelism
- Data passing overhead – Data transferring time should also be considered.

2.4 Considered Approach for Implementing Real-time SDR-base 5G systems

Short TTI and multiple antennas will be the main areas to focus on for 5G real-time SDR, according to all the descriptions above. Short TTI has the drawback of having less PHY processing time, which eliminates the possibility of successive processing. Regarding several antennas, they signify a larger data stream that must be processed separately and take more time to transport. Combining these two requirements, the 5G real-time SDR must process data quickly and have a high throughput. Additionally, as 5G has widely dispersed antenna nodes, having a potent central processing unit looks more logical.

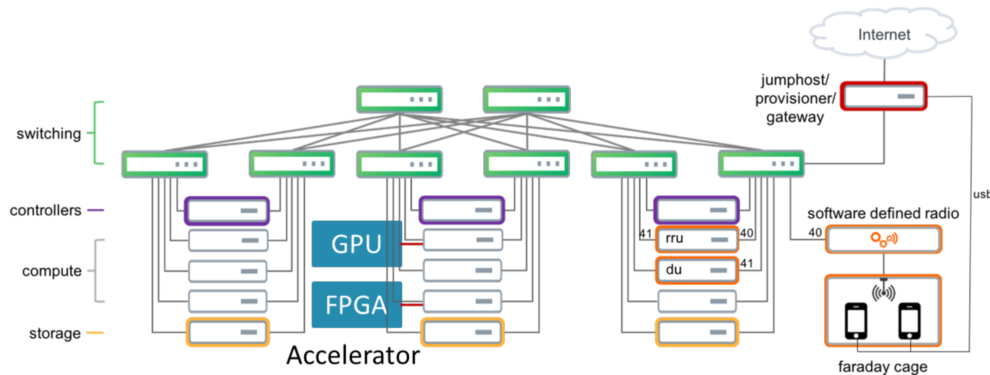

Figure 2.4 – Data-center environment with FPGA/GPU accelerator

According to the investigation, a slot time was required to finish a 4G process when a sequential process structure was used to handle a single layer of data to one antenna. It is not feasible to keep the same structure for 5G. Acceleration is unquestionably required. From the chapter above, there are a few techniques to quicken. The data-center BBU architecture is what we're aiming for, combining all these techniques. A quicker SDR with a more evenly distributed work offload is produced by connecting many processors for work distribution within the data center, as demonstrated in figure 2.4.

Accelerating every PHY step is crucial to solving problems with long execution times, especially for those on timing important paths. Employing the previously indicated technique, either by adding an outside accelerator or by using a customized instruction set for the CPU. Since some of the steps in the procedure were not dependent on one another, reconfiguring them for more parallel execution can boost performance and make the best use of available processing power.

Multiple machines are joined in the data center base station with the intention of acting as one large unit. The topic of controlling and collaborating with each machine is crucial. Prior to then, the problem was how to arrange work reasonably even with just one machine. One method of offloading involves dividing the independent process among the machine's cores to balance the workload. However, fragmenting the process, accelerating it using CPU instructions or an accelerator, and then rearrange it will make it more difficult to read the code and to continue with additional development and maintenance. The article's focus will be on striking a balance between performance and development potential.

Chapter 3

Functional Decomposition and Pipelining for 5G radio processing

For the 5G real-time SDR, structural reconstruction is required from the previous chapter. The 3gpp communication rules state that there are no restrictions across channels, therefore they can theoretically be executed simultaneously. This results in the execution of all channels concurrently and in parallel being the most timing-efficient method. However, there are still Media Access Control layer (MAC) layer settings required for the PHY channel as well as answers from the prior slot, such as ACK/NACK responses that are shared by all channels. The most computationally intensive channels out of all of them are PDSCH and PDCCH for the downlink channel and PUSCH and PUCCH for the uplink channel. The most cost-effective solution is to divide only these heavily laden channels from other channels, as opposed to running all channels in parallel. Since fewer physical threads will be used at once, but jobs are distributed among threads more equitably in this situation.

According to the previous explanation, the maximum process time for the pair is 4 TTI from the RX antenna input to the TX antenna output if slot n for uplink and slot n+4 for downlink are grouped together due to ACK/NACK response. The new structure must prevent situations where the processing time might go beyond the slot time, endangering fronthaul devices, even though Real-time SDR must collect and send data to and from fronthaul devices at every slot time. The pipeline structure is brought up into the picture. Because of the pipeline layout, the processing time for a single string can be multiplied by the number of pipe stages while still retaining the same throughput. And because it is intended to be receiver-triggered, the processing time for the pipe stage is one slot time. Given the complexity of 5G, the structure will be divided into four pipe phases, including front end processing and channel processing for both uplink and downlink.

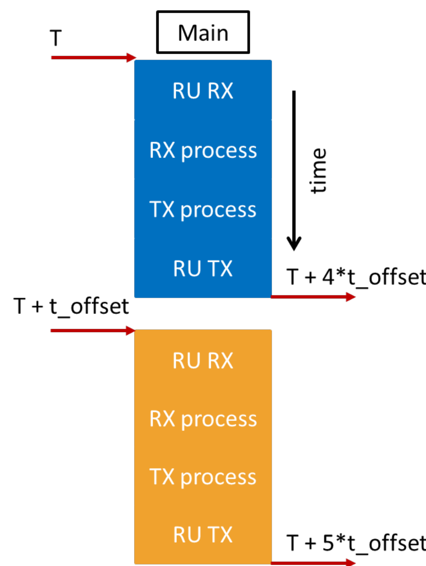

Figure 3.1 – Consecutive execution for receive and transmit set

3.1 5G Physical-Layer Procedures

Having all of the channels in 5G execute one after the other is the most natural method to build an SDR. The received signal, which the fronthaul gives out every TTI, is the only trigger that the processor does not control, therefore due to the synchronization of the fronthaul devices, the PHY process should be triggered by the received signal. Additionally, the receive signal includes the downlink's answer, and the corresponding downlink signal, which must be transmitted back four TTI later, will be configured in accordance with the response. The most fundamental structure for SDR will result from combining these two features, as shown in the figure 3.1, which has the fronthaul process for uplink, followed by the physical layer for uplink on slot n, followed by the physical layer for downlink, and finally the fronthaul for downlink on slot n+4. The processing for the following set begins after the set for processing slots n for uplink and slot n+4 for downlink is complete.

It is obvious from the previous description that the structure must have finished processing the n+4 downlink signal before receiving the following slot. Due to the fact that the uplink fronthaul is activated every slot time, both the uplink n and downlink n+4 processes are completed far in advance of when they are required. Additionally, because it is a 5G SDR with many antennas and the potential for several data layers in Layer 1 (L1), the physical layer, the amount of data being delivered has significantly risen and the TTI will be shorter. Although the structure might work for 4G SDR, which uses a time slot of one millisecond, it won't be ready for 5G SDR, which uses a time slot of five hundred microseconds more frequently but may be significantly shorter in some circumstances.

The basic structure needs to accelerate, but how to accelerate is the question. The

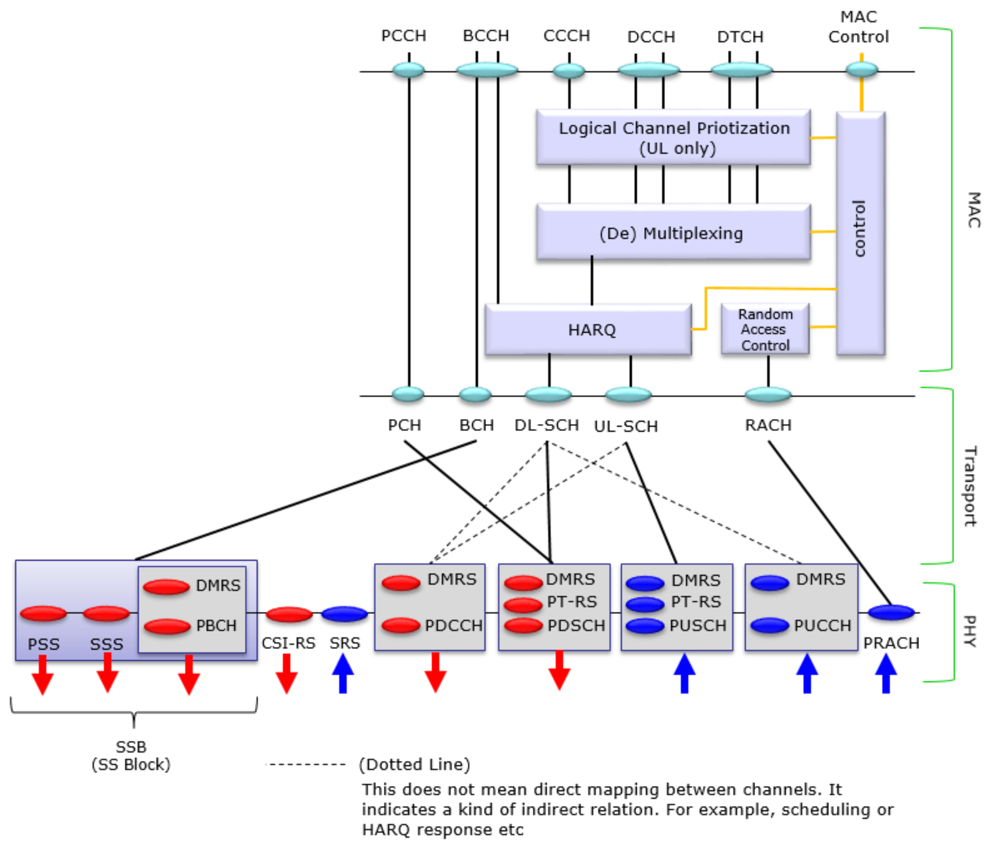

Figure 3.2 – Channel mapping diagram from ShareTechNote

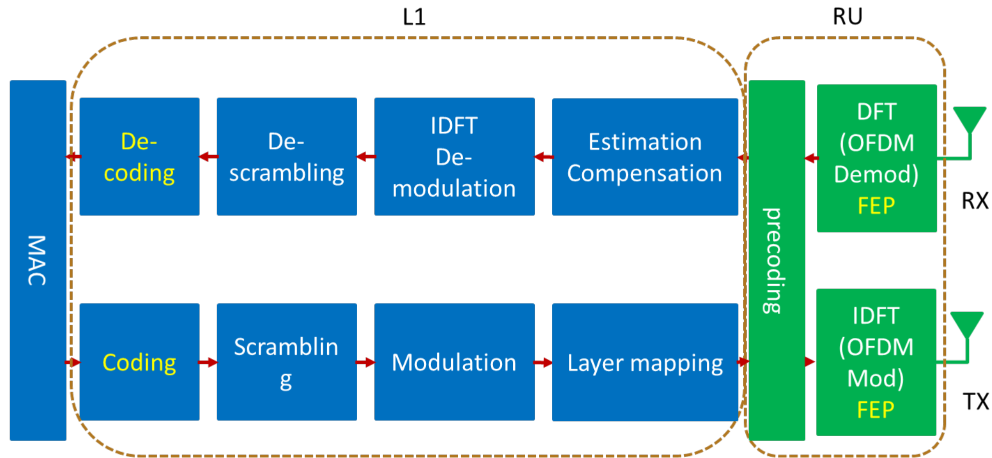

Figure 3.3 – Physical layer flow with the functional split

figure 3.2 contains a description of all PHY channels for both uplink and downlink as well as their relationships. The execution of every block for PHY in figure 3.2 represents the greatest parallelism that can be done on the SDR, in theory. Even though it may be among the fastest methods for SDR, some of those channels are short to execute, splitting it in parallel will just result in overhead due to hand shaking within threads. A sudden increase in the need for computing resources will result from splitting off all these tasks. There will be additional overhead if there aren't enough cores to support the processing because the Operating System (OS) will have to work extra to arrange and manage them. Additionally, it is more difficult to maintain and add new features for development with a shredded architecture.

Figure 3.3 illustrates the process that the majority of data in PHY must go through by approaching it from the point of view of flow rather than channels. According to figure 3.3, data must first go through certain processes, including coding, scrambling, modulating, layer mapping, and OFDM, before being sent out by the antennas for the downlink after MAC. Since uplink and downlink are essentially identical, the reverse operation is used to do the reverse procedure. All data must go through these procedures, whether it is control data or not, as long as it is not the signal that determines a channel's adjustment or measurement. A mechanism for matching layers and antennas is required since 5G may simultaneously use many layers and numerous antennas. The precoding process is in charge of mapping the layers and antennas. In Figure 3.3, for instance, the blue blocks are all based on layers, whereas the green blocks are all based on antennas. As a result, there are two different categories created in the processing flow. In OAI, these categories are divided into L1 and RU, respectively.

3.2 Proposed Pipelining Methods

It is apparent from the previous description for figure 3.1 that downlink signals are prepared four slot times before being actually sent out. However, it is handled at the

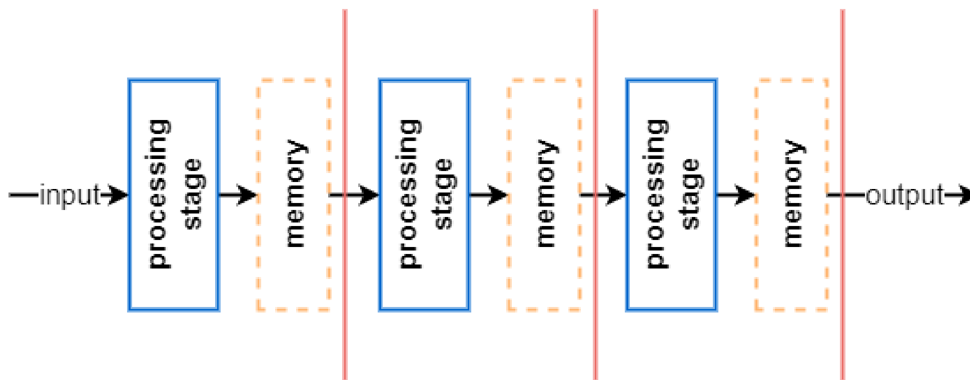

Figure 3.4 – Simplest pipeline example

same time as the uplink signal, which is much in advance of when it is required. This inspires the concept of creating a framework that keeps the antennas' ability to receive and broadcast signals at every time slot while allowing the processing time to be as long as possible. Since, as previously said, it won't be possible to create threads for each block in figure 3.2, using figure 3.3 as inspiration becomes yet another aspect for which the pipeline structure is then suggested. The process is divided into four portions by the pipeline structure, where the divisions are dependent on the uplink and downlink process as well as the L1 process and RU process. In this scenario, the procedure might be extended to four pipe stages later and still fulfill the requirement of maintaining the ability to send and receive data for the antennas during each slot despite the 5G slot time being compressed.

3.2.1 Pipeline Mechanism

One of the subjects mentioned when discussing parallel programming is pipelining. As depicted in the picture 3.4, the pipeline structure divides the entire process into segments and then separates them into several stages. After the data being handled in the pipe stage is complete, it is stored in that particular memory rather than being sent directly to the following stage since each stage has its own distinct set of memory. As a result, all pipe stages can run concurrently without interfering with one another.

A global clock in the classic pipeline layout triggers the stages to begin processing, although this may have a few drawbacks. Timing variation is the first problem that may arise from unevenly dividing the operation. Since each pipe stage has its own distinct execution, each pipe stage's execution time is likely to differ from the others. The splitting for this pipeline is unbalanced if the execution times for each stage are too dissimilar from one another. This could reduce the program's parallelism and also cause the reference data or input data for the following stage to be overwritten, as shown in the timing variation section of the figure 3.5. The second problem is data conflict, which is one of many that might arise while utilizing parallel programming. The uncertainty of who gains the first dose has a significant impact on the outcome when many writes occur simultaneously or when a write and a read occur simultaneously. As demonstrated in

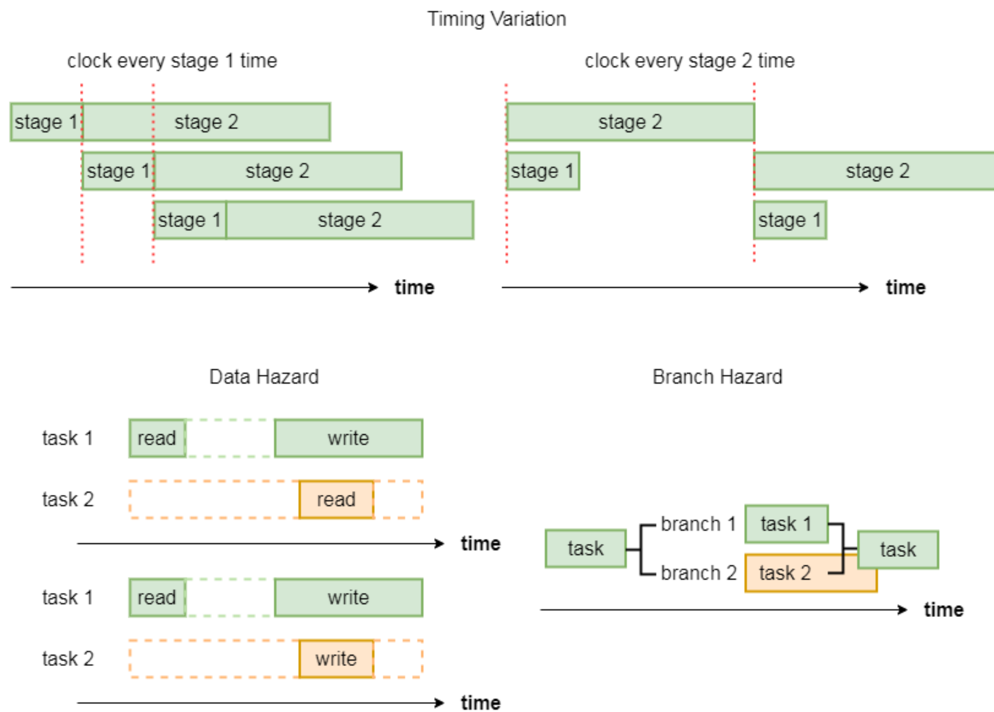

Figure 3.5 – Hazard that might happen when using parallel programming

the data hazard section of the figure 3.5, this type of data hazard will make the program unstable, which is undoubtedly undesirable in the SDR. Branch hazard is the third type of problem that could arise in parallel programming. As shown in the branch hazard portion of the figure 3.5, if the program branches out to have more than one path and then merges those paths back together to continue the process, it might have some data that is not yet ready to be handled depending on when the merged process is being executed.

3.2.2 Software Pipeline

Due of their ability to run simultaneously, threads are commonly used in parallel programming. Each thread will represent one stage of the pipeline by incorporating it into the software. How to handle shared data becomes crucial because most of the time, communication between threads is required. The mechanism we selected for implementation should be able to prevent data conflict because communication across threads may do so. When implementing, there are a few options. One of them is to use a volatile variable, which provides the compiler the impression that the value is only being changed by the present program. Another one is the atomic operation, which verifies if a variable has been altered by one action as it is being performed. The final method is to use a mutex, which is a blocking action that guarantees that only one operation can access a variable at a time. In this situation, utilizing a mutex can ensure that the variable is processed in

```
start_meas(&gNB->ul_indication_stats);
pthread_mutex_lock(&gNB->UL_INFO_mutex);
gNB->UL_INFO.frame     = frame_rx;
gNB->UL_INFO.slot      = slot_rx;
gNB->UL_INFO.module_id = gNB->Mod_id;
gNB->UL_INFO.CC_id     = gNB->CC_id;
gNB->if_inst->NR_UL_indication(&gNB->UL_INFO);
pthread_mutex_unlock(&gNB->UL_INFO_mutex);
stop_meas(&gNB->ul_indication_stats);
```

Figure 3.6 – Example of One of The Mutex Used in SDR

the desired order while also ensuring that there will be no data conflicts. An illustration of the use of mutex in the code may be found in the picture 3.6. Wherein we first log the timestamp before to the operation, and then we mutex-lock the thread. After which storing the shared variable that was sent to the function in a local variable. Finally, we unlock the thread using a mutex, log the time, and calculate how long was spent on the mutex.

Despite the fact that the first pipeline structure was suggested for the hardware design, the previously mentioned problems will still be a problem after the pipeline structure is integrated into the software application. However, there are benefits to software pipelining because all of these problems have a simpler software solution.

Data hazards can be avoided by utilizing mutex locks to grant privileges to the task that is currently handling the data. Similar to the example in figure /reffig:mutex, the thread locks shared memory, writes data into it, and then unlocks the memory when it is completed writing. Additionally, individuals who want to request writing must wait for the memory to be freed before competing for the opportunity to write data. As a result, mutex locks act as the threads' data protectors, preventing any potential data hazards.

Regarding the branch merging issue, the most crucial thing is to ensure that the following job, which should occur after the merge, only begins to run when all branch tasks before to merging are completed. The binary mask method is the one employed in the thesis to represent each branch's state. There shouldn't be multiple executions on one branch within one pipe stage because a pipe stage can only be woken up again after its present duty is complete. Because of this, representing one branch with one bit is sufficient. Bit sizes that are at least equal to the number of branch jobs are contained in the bit mask. The mask is shared by all branches, and each component of the mask stands for the associated branch. Once the branch has completed its duty, it will change the matching bit in the mask to one to indicate that it is complete. Every branch will check the mask, and if there are no zeros remaining, it is the last branch to finish, at which point the next job will be woken up. When filling in the mask, the branch hazards are dealt with using the same technique as was previously discussed for data hazard prevention. In order to prevent any missed track branches, this means that only one

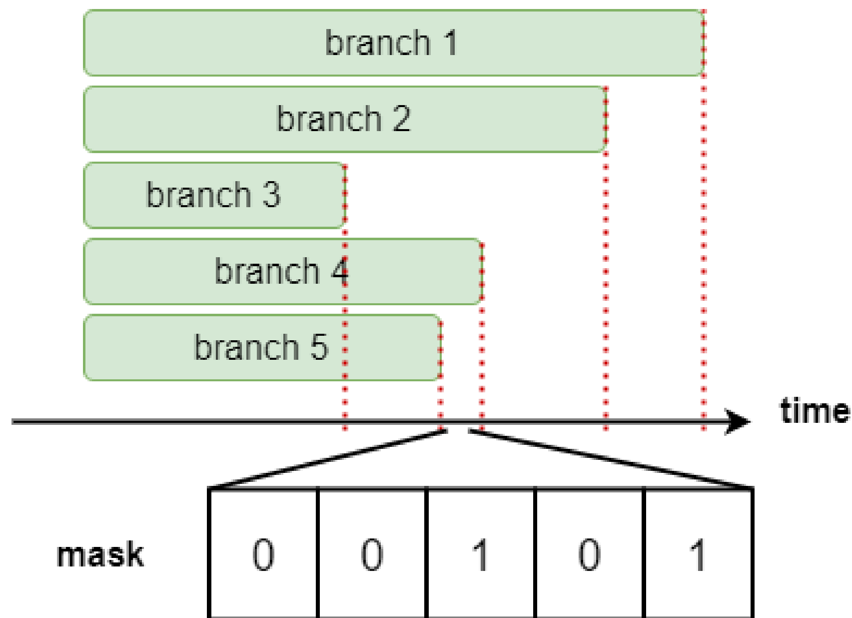

Figure 3.7 – Bit-wise mapping mask indicator

mask can be filled at a time.

The signal sent in the software pipeline replaces the clock trigger for the hardware pipeline. This indicates that rather than being woken up by each clock tick, the software pipeline's pipe stage is woken up by its previous stage. All stages will enter sleep mode right away after being established to ensure that they all follow the pipeline order. Following that, those stages continuously check for a wake-up signal from the previous stage. Each stage completes its task, wakes up the following stage, and then returns to the waiting mode to await the next signal to wake it up.

As was previously established, those threads must speak to one another. They can communicate with one other in various ways, one of which is depicted in figure 3.8. To achieve communication, mutex and condition variables are being used. It will enter the wait state if another thread is required to wake it up, and it will accomplish this by first locking the condition variable and then doing the waiting on it. Once the condition variable has updated, the thread will resume its activity. The thread signal, which is used to wake up a thread, is sent when a thread wants to wake up another thread. The other thread will then receive the signal that it is time to wake up and check the condition variable to see if it matches the requirement after the thread locks the condition variable and modifies it to fit the condition necessary for it. The lock is then released by the thread so that other threads can do the action on the same memory. The thread that has completed its task and is waiting for a new signal to tell it when to begin the next round of work will lock the condition variable, modify the condition variable back, unlock the variable, and then resume waiting.

When all of these features are combined with a software pipeline, all pipe stages other than the first pipe stage have to enter a wait state after the thread is created. The

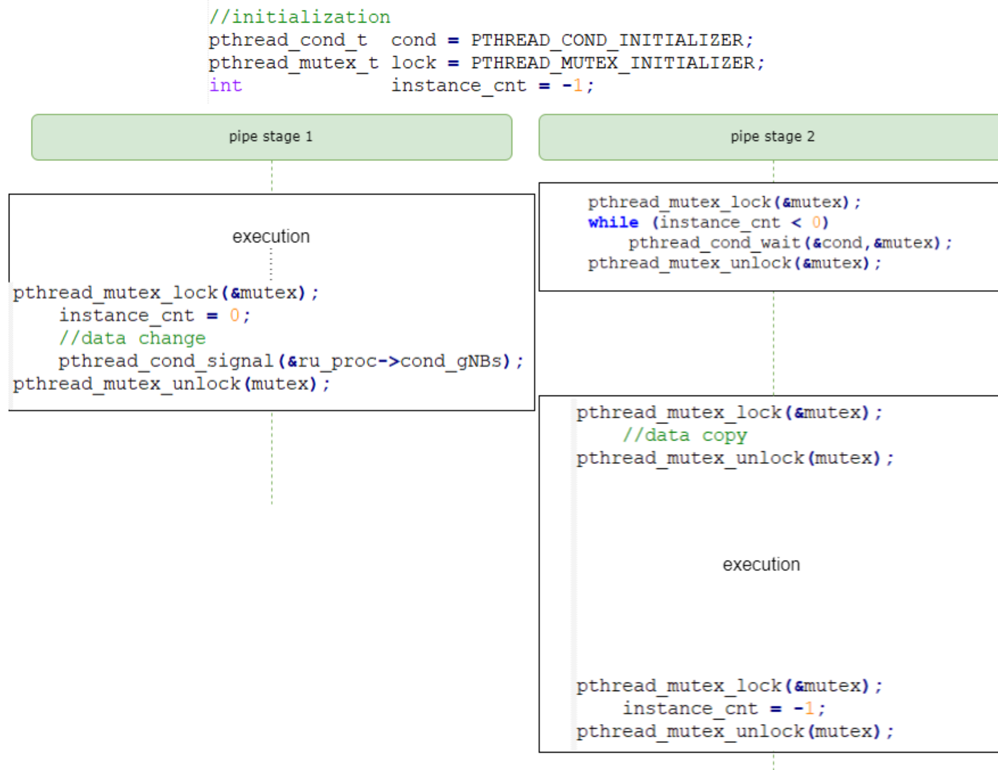

Figure 3.8 – Threads communication with mutex lock introduced

frequency at which those pipe stages can be triggered is determined by the first pipe stage. The difference is that the next move is now started by only its previous stage and not by each pipe stage clock as before. This could lead to jamming because each pipe stage has a different execution time, and jamming typically occurs at the stage that takes the longest to execute. Make sure that no stage's execution duration exceeds the time for the frequency of the trigger from the first pipe stage to avoid jamming, which was thought to be one of the effects of timing variation problems.

3.2.3 Proposed Pipeline Structure

Downlink signals for n+4 are generated using the original structure after uplink signals n are received. These downlink signals are, however, sent four slot times later. This indicates that the new structure just needs to produce the relevant downlink signals within four slot times. Pipeline approaches are introduced to provide these characteristics while retaining the ability to receive and send at every slot time and extending processing time for the physical layer processing set to four slot times.

The SDR pipeline topology has the benefit of a better throughput by utilizing overlapping stage execution. However, if we divide the operation into too many steps, the latency will increase because of the time spent communicating between threads. Furthermore, because to the high complexity of the thread, if there are too many threads

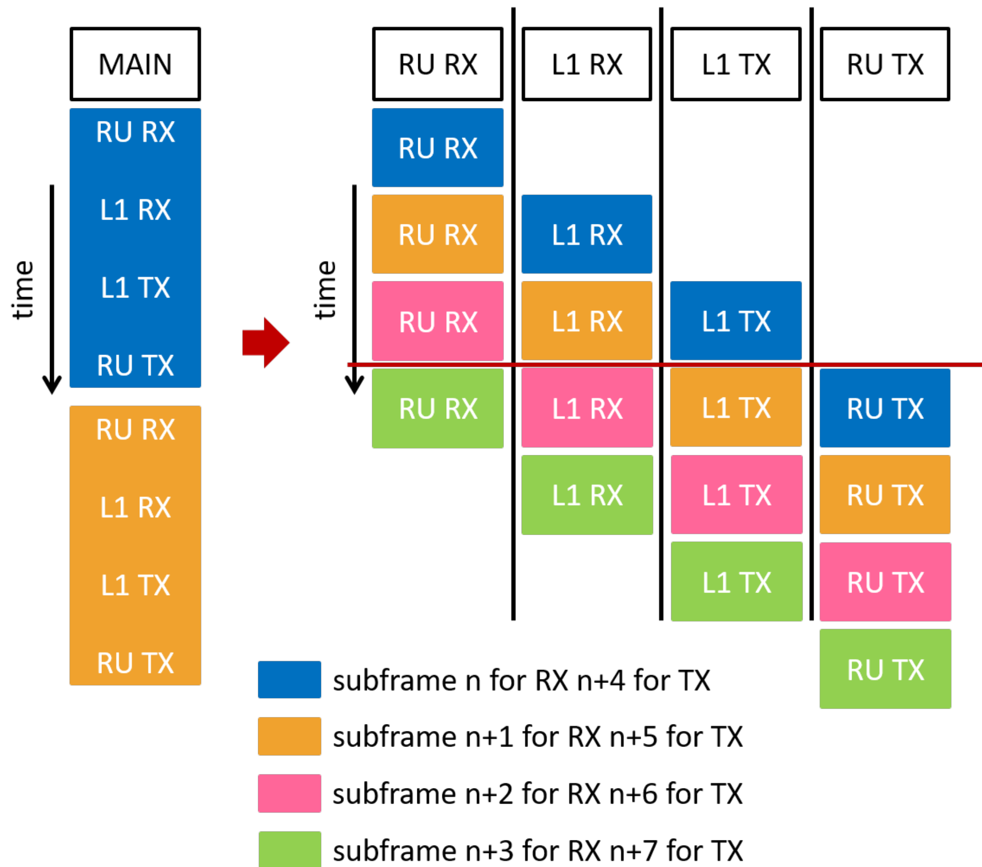

Figure 3.9 – Pipeline structure works in timing aspect

utilized in the program, it will be difficult to continue developing and maintaining the SDR. This thesis achieves 5G by converting the SDR into a pipeline topology to generate sufficient throughput. We can build a pipeline structure for real-time SDR by including all the ideas from the previous section into the SDR. Even division is necessary, nevertheless, in order for the SDR to escape the time variation problem brought on by pipeline structure.

Speaking about balancing splitting, it is preferable to have the splitting as indicated in the figure 3.9 due to the distinct bases of the L1 and RU, which are layer and antenna. The most logical solution is to divide into four pipe stages since the set of the receive and transmit pair includes the transmit for four TTI later. Because of the pipeline arrangement, the receive and transmit pairs are not required to complete in a single TTI. The timeline shown in the figure 3.9 clearly shows that all pipe stages are processing simultaneously, but the SDR can still have the incoming and outgoing data ready at each slot time, and from the perspective of the receive and transmit pair, they now have four TTI instead of one to allow them to complete their process.

Chapter 4

Acceleration Methods for 5G Functional Blocks

Since functional acceleration will be covered in this chapter, it would be wise to look at how those functions work. According to the standard, the bandwidth, sub-carrier spacing based on numerology, and modulation coding scheme (MCS) all play a role in how all data is handled in SDR for 5G. After these factors are chosen, the amount of the bit stream data—which is equal to the number of resource blocks multiplied by the modulation scheme—is chosen. This is illustrated in Figure 4.1. More data will be carried by single resource blocks when the MCS increases since the data encryption rate increases along with it. The SDR will take longer to operate the longer the bit stream data is. The heaviest operation loading in time should be manageable by a 5G real-time SDR. When the SDR is heavily loaded, it is easier to determine which function requires more computing time and resources than other operations.

Despite the fact that the previously specified pipeline configuration for SDR. Even yet, a few operations still need more time than others, particularly when the high data rate is in use. The processing time for one slot per pipe stage will still be exceeded by the pipeline structure as a result of these lengthy functions. The question of how to speed up function execution is the one that will have an impact on the pipeline structure's stability.

4.1 Acceleration Target Choice

As already noted, in order to match the throughput of 5G, all of those pipe phases must be completed in a single slot time. On a general purpose processor, the SDR pipeline arrangement without any acceleration is still insufficient. Acceleration over the function will shorten the execution time for the current pipe stage, which may allow the SDR to continue operating steadily under heavy data loading without experiencing traffic jams. The initial stage will be to measure and analyze the execution times for each function in order to choose a candidate for the functional acceleration.

Each block in the block diagram of figure 3.3 in previous chapters, which serves

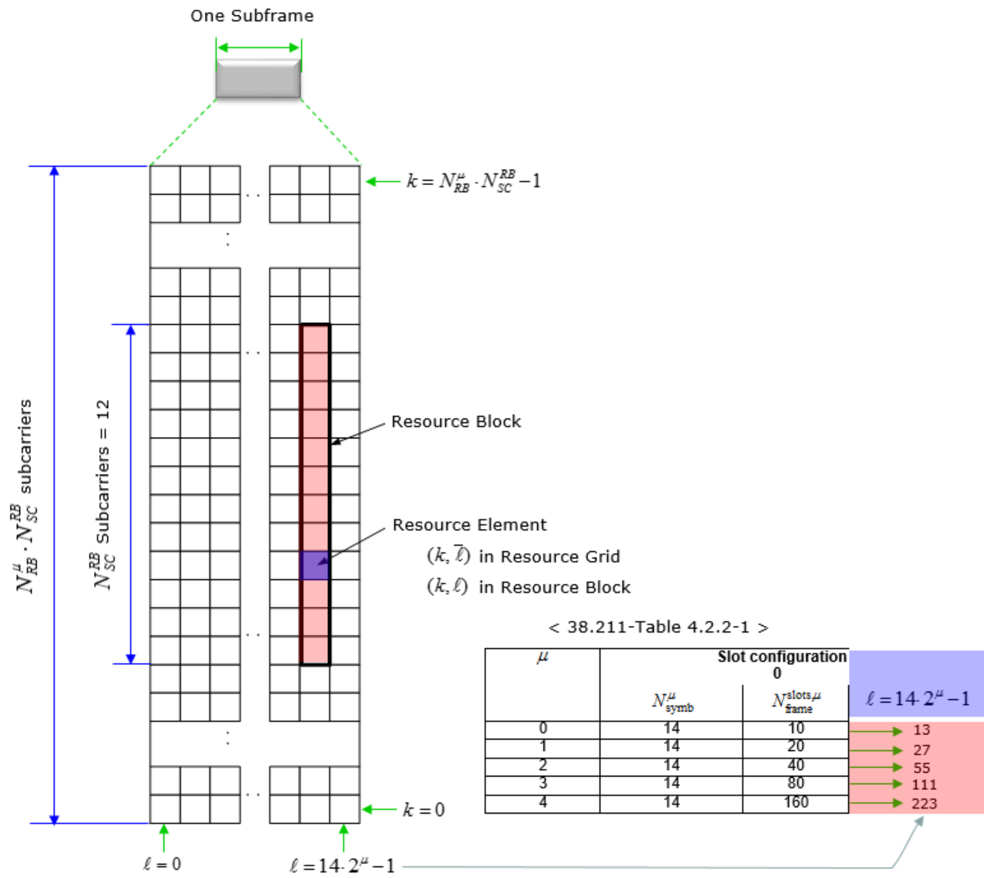

Figure 4.1 – Resource grid for NR from ShareTechNote

as the minimum accelerate unit in this case, represents a distinct PHY process. The target that might be selected to accelerate is also depending on how each step must be completed before the next phase can begin. This section discusses the acceleration for the functional blocks of coding, scrambling, modulation, and IDFT for downlink, and decoding, descrambling, demodulation, and DFT for uplink, which is the opposite of downlink. The key issue will be how to speed up each step's execution. Rearranging the process and shortening the execution time are the two methods available for carrying out the task. Rearranging the process may involve employing an accelerator or threads to perform parallel processing.

Coding is the initial block to enter the site. In replacement of the turbo encoder technology used in LTE, Low-Density Parity-Check (LDPC) is the coding technique used in 5G. The parity check bit string is generated by the original data passing through a sparse matrix, and it is attached to the end of each encoded data block as part of the LDPC encoding process. The 3gpp standard includes LDPC, which supports several spars matrix sizes. The broader the matrix size, the longer it takes to process a single input string since there are more parity check bits to generate. The following equation represents the LDPC operation.

$$H = (PI), I = identity$$

$$inputstream : m = [m_1, m_2, ..., m_n]$$

$$paritycheckstream : p = [p_1, p_2, ..., p_x]$$

$$codedstream : c = [m_1, m_2, ..., m_n, p_1, p_2, ..., p_x]$$

$$Hc^t = 0$$

One example of LDPC encoding uses a block size of six and three parity check bits to illustrate the concept of LDPC coding more clearly.

$$P = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

$$H = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 1 & | & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & | & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & | & 0 & 0 & 1 \end{pmatrix}$$

$$m = [m_1, m_2, m_3, m_4, m_5, m_6]$$

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \\ m_6 \\ p_1 \\ p_2 \\ p_3 \end{pmatrix} = 0$$

$$p_1 = m_1 + m_3 + m_6$$

$$p_2 = m_2 + m_5$$

$$p_3 = m_1 + m_4$$

The previously mentioned example demonstrates how the LDPC creates the parity check bit; the quantity of parity check bits equals the matrix's depth, and the quantity of input data is equal to the number of rows in matrix P.

Since there are two Base Graphs (BG), measuring 46 by 68 for BG1 and 42 by 52 for BG2, the matrix in the 3GPP standards for 5G can be quite large. And since those BGs have a maximum expansion of 384, each BG number will become a bitwise matrix with a size equal to the expansion factor times the expansion factor. For the decoding, the same matrix is used, but the bit string is estimated using the log-likelihood ratio [27] using the following equation.

$$L_x(x) = P_x(x = 0)/P_x(x = 1)$$

Coding and decoding are block-based procedures, as the explanation above indicates. Each block stands alone from the others, which results in a very repetitious and data-independent action when using blocks. Since LDPC always uses big parity matrices, it will take a long time for the matrix operation to complete one block. These factors combine to make LDPC an excellent choice for functional acceleration.

By scrambling data, it is possible to lessen the chance that nearby data may become polluted simultaneously, which could make it more difficult or impossible to reverse the information. These are accomplished by having the current bit cross-relate with a certain set of bits that have previously passed through the scrambler. It received the initial sequence to work on the scrambling after passing the golden random-like sequence. Its natural successive execution makes parallel execution undesirable because the input is a bit string and every output bit is influenced by the previous input bit.

The use of quadrature amplitude modulation (QAM) is the following. A group of bits are matched to a certain set of amplitude and angle, where the type of QAM determines the amplitude and angle. The two most important factors in choosing the type of QAM are channel quality and signal dependability. Because there are fewer locations in the diagram, there will be less chance for error even if the channel state is really bad when data needs to be encoded and demands high reliability. However, it usually uses a higher QAM level when the channel is clear and there is a lot of data to send since it can fit more information in a given set of amplitude and angle. The level for QAM could start at the smallest one-to-one modulation rate, like Binary phase-shift keying (BPSK), to transport multiple bits with a single set of amplitude and angle as illustrated in the figure 4.2. The string data will be aggregated and then matched to the angle-amplitude set after the level of QAM has been chosen. As a result, there is no connection between the groups, which makes parallel processing of QAM operations simple.

Then comes OFDM. Prior to delivering a signal to the antennas, it is utilized to fix the previously described ISI and delay spread issues. Multiple orthogonal subcarriers are implemented in OFDM to transport data, and the Fourier transform is used to do this.

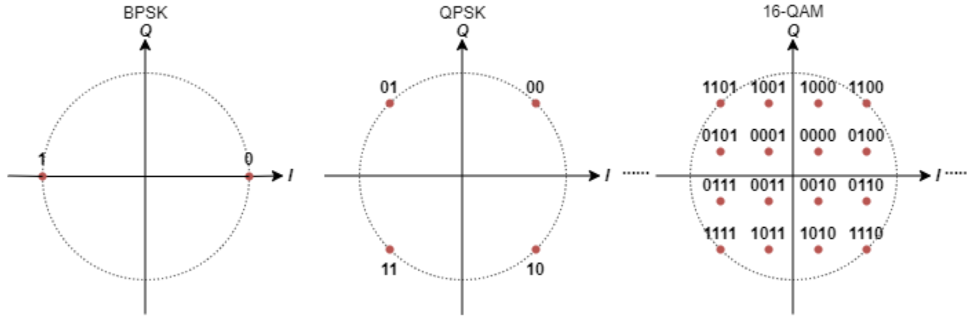

Figure 4.2 – Quadrature Amplitude Modulation (QAM) diagram example

Figure 4.3 displays the represented image. Fast Fourier Transform (FFT) and Inverse Discrete Fourier Transform (IDFT) in 5G are capable of processing up to 4,967 points, which means that the operation will involve a 4,967 by 4,967 matrix. The following shows how the equation will appear and an example of FFT and IDFT with n points.

$$FFT: \begin{pmatrix} P(x_0) \\ P(x_1) \\ \vdots \\ P(x_{n-1}) \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_{n-1} \end{pmatrix}$$

$$x_k = \omega^k, \text{ where } \omega = e^{\frac{2\pi i}{n}}$$

$$\rightarrow \begin{pmatrix} P(\omega^0) \\ P(\omega^1) \\ \vdots \\ P(\omega^{n-1}) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_{n-1} \end{pmatrix}$$

$$\text{for IDFT} \begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_{n-1} \end{pmatrix} = \frac{1}{n} \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} P(\omega^0) \\ P(\omega^1) \\ \vdots \\ P(\omega^{n-1}) \end{pmatrix}$$

It is clear from the equation above that the process has an $O(n^2)$ level of complexity. There are techniques to make the operation simpler, however even with a better algorithm, the complexity will still be $O(n \log n)$. The data string is then divided into groups for the operation since IFFT and IDFT have a maximum point. The link between the time domain and frequency domain carried out by the Fourier Transform is also depicted in Figure 4.3.

As was already established, scrambling is the only activity in real-time SDR that cannot be sped up by splitting into groups. The acceleration for function needs to be chosen from those operations demonstrated before since it can result in timing problems or data jamming as previously described owing to having an excessively long execution

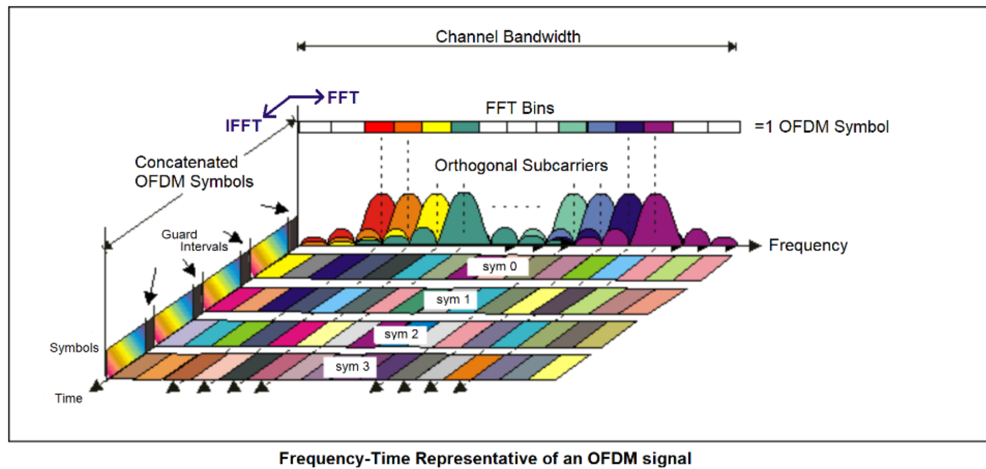

Figure 4.3 – OFDM effect on resource element for both time and frequency domain from 5G Technology World

time. Coding and decoding come in as the top contenders. They demand a lot of time and resources. Even after testing all these functions in the simulation, they end up costing the majority of the time. It is the ideal candidate for functional acceleration due to its block-based, matrix processing, and lengthy execution time. As for the following prospective QAM. It can be parallelized as well, but all it does is map a bit to an angle-amplitude pair, which is set up quickly. It may require more time to accelerate it using reconstruction or an outside accelerator because the overhead is greater than the acceleration gains. This indicates that QAM functioning isn't well suited for acceleration. Since multiple antennas are utilized in 5G SDR, having all of the data for various antennae execute sequentially is inefficient for FFT and IDFT. Both FFT and IDFT contain matrix operations that are highly complex and independent when used with various antennas. They are strong candidates for acceleration due to all of these.

4.2 Splitting Method

When the additional acceleration device is not connected to the PC, one of the accelerating approaches is to use additional threads as the workers to offload the initial operation. Functional threading parallel is used to process the data and collect it at the end of the process by having one dominant thread that manages multiple worker threads. There won't be any dependency within groups for those that have group operations, making it simple to split them up. The function that consumes the most time during the pipe stage is given priority when it comes to splitting.

We want to use worker threads in the real-time SDR to divide the workload among the most time-consuming operations in the pipe stage. Coding and decoding are the PHY processing bottlenecks, as was previously mentioned. It is evident that coding and decoding are block-based processes by looking at how the data is handled in the function.

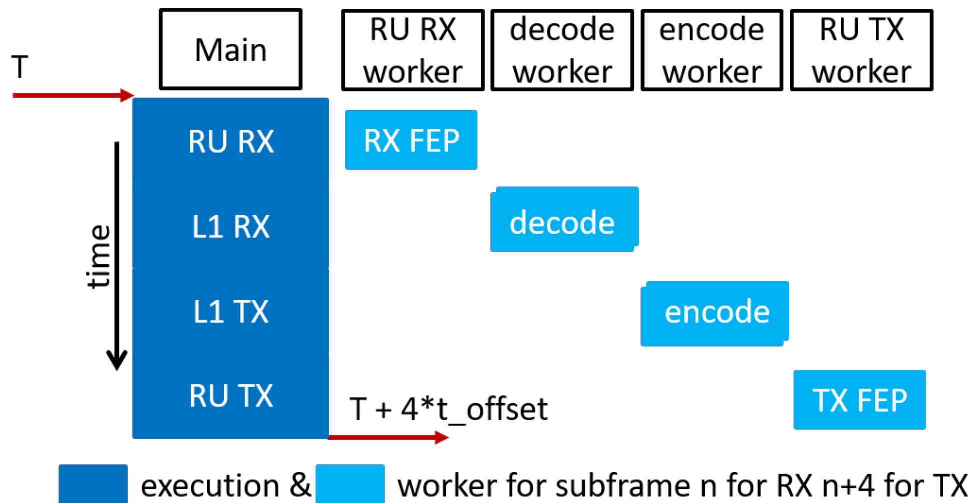

Figure 4.4 – Including worker threads to the procedure

Because of this, we must divide some of the blocks into worker threads for offloading. Prior to the main operation, the main thread handles the common operation and then distributes some of the data to the worker threads that have identical tasks so they can run in parallel. The total execution time will be made up of the time for common operations, the time for operations with fewer groups (calculated by dividing the original execution time by the quantity of worker threads), and the time spent waiting for each worker thread to complete its own task and signal the main thread that the execution is complete. The timing of common execution time with a portion of the parallel process execution time, in theory, results in the best performance.

The data handled in the RU pipe stage grows by the original time number of the antennas since 5G uses many antennas. Both the DFT and IDFT procedures operate on blocks, and data between various antennas is independent. As a result, sequential execution of antenna works is ineffective. Since each piece of data that is given or received has a timeframe, the structure may prevent the data from being generated in the allowed period of time. One method of performance is to parallelize tasks over antennas. utilizing the appropriate threads to serve the IDFT and DFT operation for each antenna, and sending the data to the fronthaul to enable the antenna to broadcast the signal. By doing this, the time required for the original operation times the number of antennas is reduced to maybe somewhat longer than the original operation time due to overhead. The execution while adding worker threads to the original flow is shown in the figure 4.4.

When the pipeline structure and worker threads are integrated into the SDR, the architecture will resemble that in figure 4.5. The four-pipe stage is designated here as RU RX, L1 RX, L1 TX, and RU TX. The task of OFDM demodulation, or DFT, for each antenna is then offloaded to worker threads when RU RX has received the data. After each worker thread has completed its task, it should fill in the corresponding bit in the mask. A message will be returned to the RU RX stage by the final worker thread

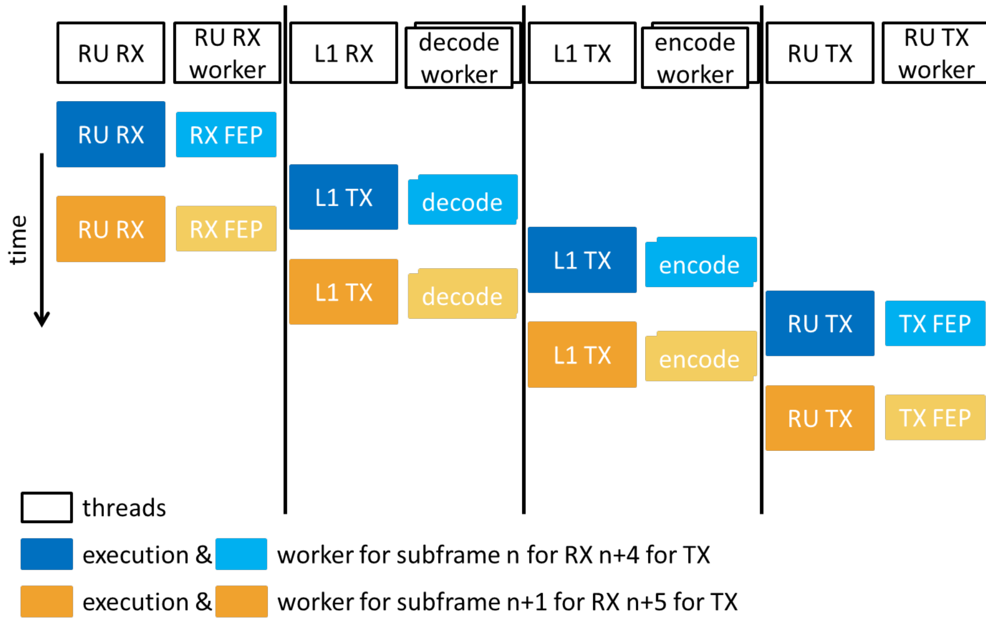

Figure 4.5 – Combing worker threads and pipline structure

that completed processing. To ensure that all worker threads have concluded, the main thread—RU RX pipe stage—will verify the mask after that. The RU RX stage will then take the following action to awaken the L1 RX stage if all the worker threads have successfully completed their tasks. Similar to RU RX, L1 RX offloads its decoding work to the worker threads. The worker threads in the L1 TX stage and RU TX stage act similarly as well.

Splitting the process to an external accelerator, like an FPGA or a GPU, is another technique to approach the acceleration. Before transmitting data to any outer accelerator, the data should be restructured into block-wise form because the outer accelerator typically has its own method of data arrangement. The data could only be sent to the devices for parallel processing after that. After the process in the outside accelerator finishes, it's time to collect data. The data should then be returned to the pipe stage by the outer accelerator. Be advised that any data supplied back may require additional reorganization for the following process.

When using the outer accelerator, there are more data traveling, thus we should concentrate on the operation that takes the longest. Coding and decoding took up the greatest time and resources throughout the PHY operation, as was previously explained, thus they will be the target of this form of acceleration. The use of GPU as an exterior accelerator will be discussed in the paragraphs that follow. The inbuilt parallel programming capabilities of the GPU offer it an advantage when it comes to quickening a replicating operation. And because of this, GPUs are a solid choice for coding and decoding. However, as was previously mentioned, all the data needs to be reorganized before being sent to the GPU for processing. Data flowing to and from GPU

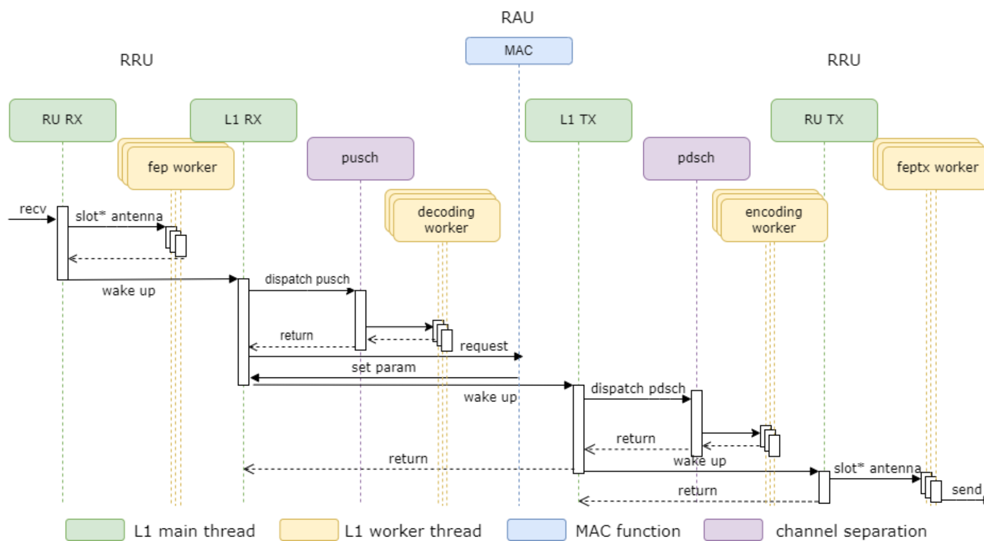

Figure 4.6 – The communication between threads when combining pipeline structure and worker threads

causes a significant latency, especially at the data center base station. This indicates that using regular buses is insufficient for data transport. A 5G data center will require buses with high throughput, greater than 1 Gbit/s, in SDR. All of these imply that latency caused by data transfer must also be taken into consideration. While GPU offers native support for parallel programming, it also has more processor groups, simpler cores, shared memory inside the processor group, and slower clock rates. All of these make GPU programming more challenging to design and require more parallel design work.

The combination of worker threads, pipeline architecture, and accelerator devices is depicted in Figure 4.6. wherein the worker's execution may take place not just on a regular PC but also on an accelerator. The architecture becomes more complex when pipeline structure and worker or accelerator are combined, as shown in Figure 4.6. This implies that adding new features will be more difficult and that maintenance work will take more time.

4.3 SIMD optimizations

Single Instruction Multiple Data, or SIMD, is an abbreviation that already implies native parallel processing. Parallel processing is being used at the instruction level. The following example shows how SIMD is used to process many variables simultaneously.

$$x[0] = a[0] + b[0], x[1] = a[1] + b[1] ... x[n] = a[n] + b[n]$$

We can infer from the equation that the operation simply adds each element of "a" and "b," storing the result in "x," which can then be used in the vector process "X = A + B." This is where SIMD happens and this is how SIMD works. A bunch of data

performing the same action, in this case a vector add, is handled by SIMD. By converting a loop action with basic execution into a vector operation, SIMD speeds up processing. The loop over instruction execution time is reduced to one SIMD instruction time as a result.

There are several instances in signal processing when processes are built on blocks and are repeated. Due to this, SIMD is useful in order to drastically speed up the procedure. In signal processing, there are many operations that are bitwise-based. The matrix operation LDPC, for instance, can be divided into various vector processes.

There are many benefits and drawbacks of SIMD. The biggest advantage we experience is a large reduction in operation time, which is crucial for real-time SDR. However, SIMD still has its limitations. It is a particular type of special instruction set that targets only particular cores, hence the SDR will need various SIMD instruction sets that correspond to different platforms. Only basic operations are supported by the SIMD instruction set, therefore if an operation is too complicated to deconstruct, SIMD cannot be used to complete the execution. Additionally, SIMD has restrictions on the size of the data that it can process. This implies that in order to operate, a vector operation that is longer than the SIMD data size must be divided into many data that might fit in the SIMD data size. Compared to standard C code, SIMD is more similar to assembly code. This implies that it is less automatic and more difficult to comprehend. Additionally, data will need to be transferred into the unique register needed for the process' execution. Furthermore, the data must be the power of the number in the group because the data are based on groups. Finally, because it is more like assembly code than regular C code, changing platforms is challenging because of how environment-dependent it is.

Chapter 5

Implementations Using OpenAirInterface

This chapter will demonstrate how the SDR gains from the new organizational structure. Now that the pipeline layout and acceleration techniques have been implemented, the OAI SDR can match the throughput requirement for 5G in real time.

OAI is a real-time SDR that use USRP as the frontend, as was previously stated. Different general-purpose operators alone with various USRP sets are used to test the structure. Nevertheless, it is not as simple just starting the program once it has been created, like a typical program does. OAI can only be used in a few limited scenarios, such the Linux operating system and a real-time kernel are required for the real-time SDR OAI to function. There are specifications for the CPU as well. To support the performance, the CPU utilized in real-time OAI SDR needs to have high-frequency scaling and the hyper-threading feature turned off. Since the OAI SDR requires the SIMD instruction set, the platform's CPU must be up to date in order to handle the instruction set, such as AVX256. As a result, our 5G SDR cannot use any prior releases of the CPU. OAI has already been proven to work on devices with Intel Core i5 or higher after the third generation, Intel Core Xeon after the second generation, and Intel Atom Rangeley, E38xx, and x5-z8300 processors. OAI SDR, which even required additional cores in the most recent OAI, is projected to take up the majority of at least one CPU core because OAI is a program that requires a lot of processing power. Since an SDR still requires an operating system to maintain the system, real-time OAI SDR is difficult to run on computers with fewer cores. Four cores are the minimum number of processors needed to enable real-time 4G OAI. Additionally, real-time 5G processing for OAI requires considerably more resources.

It is crucial to investigate the interaction between such a demanding process, OAI, and the OS. As seen in figure 5.1, the operating system (OS) is in charge of directly managing hardware behavior within the computer and is regarded as a bridge between software and hardware. When programs are running on the computer, the OS manages the machine's resources, such as memory. How to organize and use OS resources becomes crucial once multiple processes are running concurrently. More resources are needed in order to match the real-time performance. Additionally, since OS allocates resources

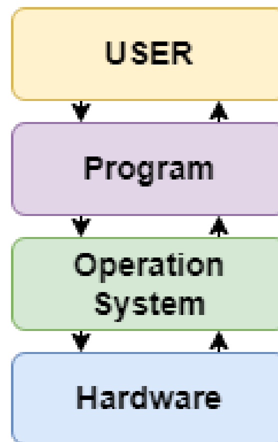

Figure 5.1 – The role that Operation System (OS) play

based on task priority, it is vital to give SDR a greater priority than other programs. Another possibility for meeting the performance requirements for 5G is to pin select threads with demanding loading operations to specific cores in the real-time SDR. By manually allocating tasks to various cores, the workload can be distributed more evenly, preventing some cores from carrying too much of the load while underutilizing others. This also helps the OS schedule jobs more efficiently. However, there are limitations for pinning operations to certain cores. Due to the possibility that various systems would have a different amount of cores and resources than the one on which the structure was tested, this may affect the SDR's ability to be accommodating when it is implemented.

A second method of optimization, in addition to managing job distribution, is through the use of compilation parameters. Here, OAI SDR is compiled using the GNU compiler collection (GCC). There are compilation choices that perform optimization with a trade-off in terms of space-speed, such as unrolling loops or inlining code. However, some optimization strategies could result in code reordering, which might create risks for the SDR because the timing of each data and the ordering in which the operations are performed are essential to the SDR. As a result, there are fewer optimization options available when compiling the SDR, ensuring that no reordering was brought about by the process. Where the majority of the optimization must be performed manually and meticulously stated in the code.

As we previously discussed, OAI was created initially for the 4G LTE framework, which has different PHY properties when importing to 5G NR. One of the few modifications is that the scheduling response for 5G NR is no longer rigidly limited to 4 slot times and is more flexible, allowing multiple slot times in different modes. Additionally, while multiple antennas or even layers are only optional in 4G, they are native to 5G. With a multi-antenna structure, more processing work is required since the frontend will need to handle multiple sets of data that all relate to the same string in a layer-based bit-wise approach. Regarding the multi-layer structure, it indicates that more than one string of data will require processing using a bit-wise operation. In addition to these, 5G has a broader bandwidth and runs at a faster frequency rate, meaning it has more resources

to process in the same amount of time whereas the analog signal has less penetration over an obstacle, resulting in a narrower coverage area from the frontend node. This forces the 5G platform to offer a feature that has a widely dispersed frontend node with a centralized processing unit rather than a few potent nodes. The coding and decoding method in 5G is replaced by the LDPC even though there aren't many modifications to the procedure flow. Due to many options for the size of the matrix, LDPC is a matrix operation that offers more coding rate options than 4G.

5.1 Testing Methods

The behavior, accuracy, and timing performance of such a real-time SDR are what are primarily examined. To ensure that the SDR can function and act properly, accuracy is required. The major goal of the thesis is to ensure that the processing time is short enough for the data to be handled in time because, aside from correctness, every piece of data in SDR is attached with a timestamp and has an expiration date. Unit tests are utilized in OAI just like they are in any programming development. Each operation must first pass through the unit test's verification process before being implemented in order to ensure that the implementation with code matches what the algorithm specifies. The algorithm was not altered by the structural rearrangement, so there should be no difference when testing with the unit test, and the outcome that passes the unit test should remain the same. The input and output must remain the same while the SDR is operating because data loss could occur while doing the rearrangement. The input and output change in real-time, so recording out the data string for comparison might not be a viable method. However, the unit test might offer the check to ensure that it is not impacted by parallelism by having a consistent output with the same input if it is a larger operation that just so happens to have the acceleration happens within the process. Additionally, the execution time for the unit test demonstrates how great the improvement is after modifying the architecture, providing an indication as to whether this form of reconstruction is even necessary or suitable for acceleration.

In addition to the unit test, there is also a simulation mode. As implied by the name, it imitates how a channel might operate in various scenarios and on several channels. Data flow through the process with simulated channel noises and defects from transmission to reception. After that, we could contrast the output with the input, which should have been identical in theory. By using the simulation mode, it was possible to demonstrate how these components work together within the channel as well as the restriction by deteriorating the channel's condition until the data was no longer retrievable from the receiver. Both the base station and the UE can serve as the receivers in the simulation modes, which use uplink and downlink channels, respectively. If all of these functions pass their unit tests, we could infer from everything mentioned above whether there is a problem with how these functions interact. The advantage of the simulation mode is that, as long as everything is in order, timing concerns are not necessary. The problems that can only be caused by real-time activity are eliminated by employing simulation, which just examines the channel's behavior and only involves one data string without an execution time constraint. This makes it simpler to focus on issues with how each

thread or function interacts with one another. To determine whether the settings are appropriate for real-time execution, we might also time the execution results while in simulation mode.

The real-time behavior of the OAI SDR is ultimately what the thesis focuses on the most. We wanted to ensure that the SDR operation could be carried out without issue within the 5G timeframe. Timing issues could create new concerns during real-time execution if they are added to the operation flow. Since executing everything in the correct order is not sufficient, it must also run within the timing constraints to avoid timing hazards like data jamming and delayed data. The real-time issue might not result in a sudden software crush on the SDR, but it could lead to delays that eventually stop the SDR from being able to service consumers. Connecting users to the real-time SDR can test the base station's capacity under the highest possible demand. Additionally, operating OAI in real-time while connected to users can assist in analyzing the effectiveness and correctness of the algorithms implemented in OAI. The order of execution and data processing are the key timing issues with the multi-thread design. When real-time is added to a multi-threaded architecture, the behavior is constrained, so all executions must take place in the proper order and must be completed on time.

In addition to all of the above, a tool known as VCD is also used to examine thread and function activity. It has the ability to capture the state of the threads as they run, making it possible to pinpoint the exact moment when the system started acting incorrectly. What it does is create a second thread with its own timeline that continuously logs the signal delivered by the process, which represents the function's activation and deactivation or the value of the variable. This illustrates the relationship between the functions, which is equally applicable to the relationship between the threads. It is simple to determine how long it takes for those threads to execute because it logged the active time of those threads. Despite the fact that it has a lot of advantages, VCD is a large application that uses a lot of resources that are intended for the SDR. VCD delays the timing and may also alter the behavior because it continuously monitors the values and status of all these variables and signals from the SDR. So it is preferable to implement VCD when debugging.

5.2 Performance Evaluation Methods

Performance can be characterized by a variety of factors, including timing, resource usage, and power consumption. When it comes to resources, this kind of speed enhancement seeks to use less memory and CPU. From a power perspective, the program should use as little energy as possible, which increases the durability and environmental friendliness of the electronic devices. Regarding timing, the better the program is, the less time it takes to run. Additionally, it implies that the program responds to requirements more quickly, resulting in a reduction in latency and reaction time when using the SDR. Because of this, timing performance is crucial and will be the main subject of this thesis. If a program is willing to put everything else at risk in order to increase timing performance, there is a good probability that the timing improvement will be insufficient to make up for the losses, which will make the program no longer meet the requirements for OAI SDR. The

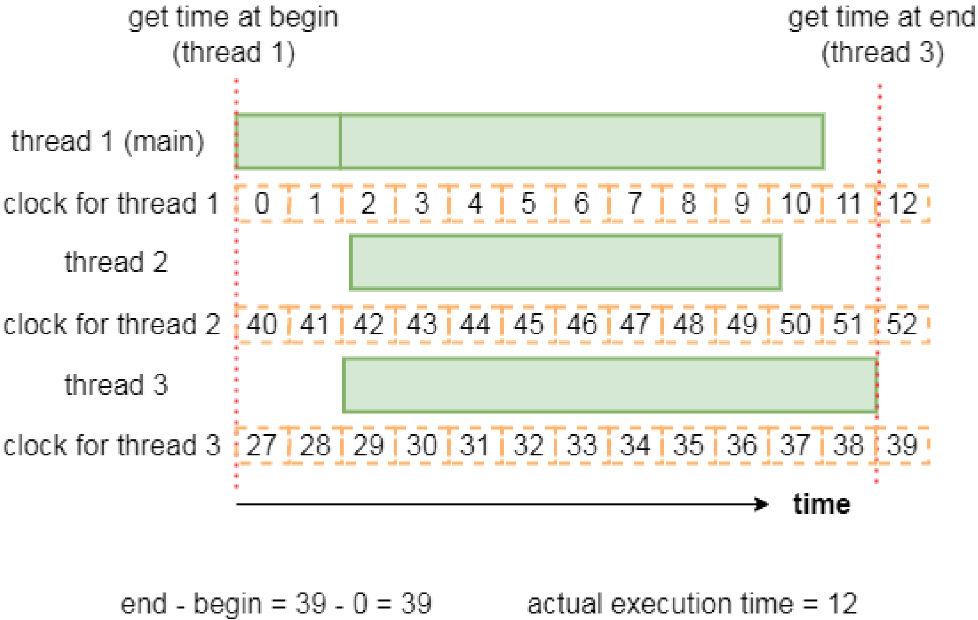

Figure 5.2 – Timing measurement on different threads

extent to which the ability to trade resources for timing will be taken into consideration, even though there is still a trade-off between time and space.

Since timing is what we are most concerned with, we must first determine the amount of time spent on each operation and the important path in the program. OAI uses the timing difference between CPU clock times to calculate how long specific operations take. Simply note the CPU time at the start and finish of the process, and then subtract those two values to get the total amount of time spent on the operation. However, since the OAI is a program with multiple tasks running at once, jobs might be distributed to different cores for the operation, there are some cases when the method doesn't truly function. Since the clock timing from any two CPUs will differ, obtaining the clock timing from several CPUs and then doing the deduction has no practical meaning in this case. This means that keeping track of each task and how it relates to other tasks is crucial in this situation. The OAI VCD tool is used to accomplish this, and it even displays the relationship with graphical plotting, which further clarifies the temporal relationship between various operations.

As was previously mentioned, there is a reasonable difference in timing between clocks on the same processor, but different processors may have different timing, so taking measurements on different cores won't give the same results for operations when running a multi-threaded process, as shown in figure 5.2. The fact that there may be some timing distortion between CPU clocks in practice only makes the situation worse.

The time difference between the start and finish of the operation can be determined in a few different ways. The first method involves using the get clock function found in the C library. A few of the functions in the library, which was developed for timing

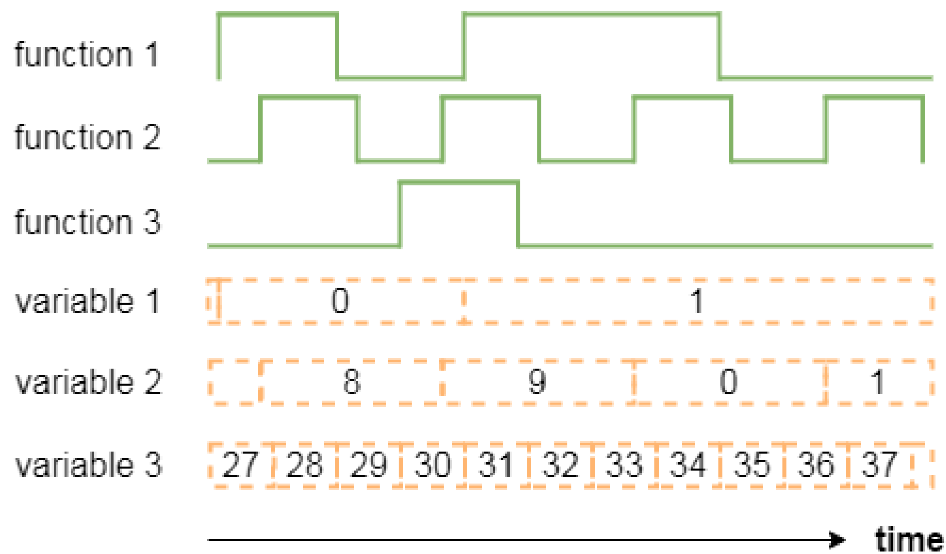

Figure 5.3 – VCD tracking for function and variable

measurement, deal with setting and accessing the clock for the running program. The execution time is determined by first setting the clock once at the beginning and then obtaining the time at both the beginning and the end of the process. Since nanoseconds are the smallest unit supported by the library, its accuracy is limited to nanoseconds. There are additional methods for determining the execution time, one of which is the timing discrepancy between the processor's time stamp and the execution time. Using the assembly syntax provided by the GCC is the way to retrieve the time-stamp. While the application is running, a trigger within the code causes the core-specific assembly code to obtain the processor time-temp counter number. The execution time of the process might then be determined by calculating the difference between the two time-stamps of the processor.

VCD is a different system that can be used to measure the passage of time, particularly for multi-threaded systems. As previously noted, the VCD generates a plot with every function and variable that was logged while the process was executing, as seen in figure 5.3. The advantage of this method is that it demonstrates the relationship between functions and variables, even though it might not be as precise as retrieving the time clock straight from the core. The plot makes it simple to determine the function's order. This means that it is simple to determine the level of parallelization for threads if VCD is used to track the activity of the threads. Despite these advantages, creating the VCD document itself takes up a lot of resources. It requires the application to deliver the signal for recording because it is executing on a separate thread. Since there will always be one fewer thread available while VCD is enabled, performance difficulties may arise when the SDR is pushed to its limit. There will also be less RAM available for the program because VCD is simultaneously writing to the file for recording.

5.3 Results of Performance Evaluation on multi-core x86 platforms

Although OAI has been tested on a number of systems and USRP sets, this test is mostly conducted on a machine with 36 Intel cores and the USRP n300 series as the frontend. As said, OAI was created in 4G with a sequential operating system and little acceleration. Its PHY activities were based on the n and n+4 receiver-transmitter set, where the subframe time for 4G is one millisecond. We begin by incorporating the 5G concept of numerology into the structure. The fundamental time unit for 5G is a slot, where numerology determines the subcarrier spacing, which ultimately results in a variable timing for the slot time due to a frequency trade-off. The consequence of the execution going beyond the allotted time has begun to become apparent after the addition of numerology to the picture. Coding and decoding are now done using LDPC, which is another significant update to the method. Figure 5.4 for numerology 0 displays the execution result for combining the LDPC method and numerology. According to the timing in figure 5.4 for numerology 0, the total execution time has already above 500us, which will result in unrecoverable execution overtime for numerology 1. Then, the previously specified features—the pipeline structure alone with functional acceleration—must be added to the SDR. Figure 5.4 is the outcome of running the SDR under the given conditions with numerology 0 and numerology 1. The SDR is still able to manage even when the signal's execution time for one slot is larger than that of one slot, as illustrated in Figure 5.4 numerology 1, from which we can plainly see the parallelism of the pipeline structure. From the perspective of the fronthaul, it can only sustain receiving and transmitting at each slot time with the pipeline topology; otherwise, late packages would pile.

As was previously indicated, employing threads to offload the task could result in latency in addition to real execution time. Two types of pipeline splitting are suggested, as shown in Figure fig:pipeline$_{s}plit, topreventtheoveruseofthreads.TheRRUandRadioAggregationUnit(RAU)ar$

The splitting is seen in Figure 5.6 from a different angle. The targets for the accelerations are three groups in the procedure flow. The first one, which takes the longest to execute according to the timing study, is bit-level processing, which includes coding and decoding. The second category includes all the other layer-based execution techniques. Instead of layering, the third component uses the antenna-based method. Combining figures 5.6 and 5.5, we can see that each pipe stage has a number of worker threads.

Let's start by discussing the worker thread used in the L1 coding and decoding method. We use eight worker threads to offload the coding and decoding operation in order to be able to comply with the 5G specification. All segments are then evenly distributed across these eight worker threads, from the perspective of coding, when the preparatory stage prior to beginning the coding method is complete. The similar approach can be used for decoding. According to testing with various SDR setups, if worker threads are not enabled, there will be a lot of late package signals from the fronthaul, which will cause the program to crash. Each piece of data must be ready on time because each of them has a timestamp that corresponds to it. Data jamming will happen if the processing power required to perform those demanding loading processes was insufficient due to a

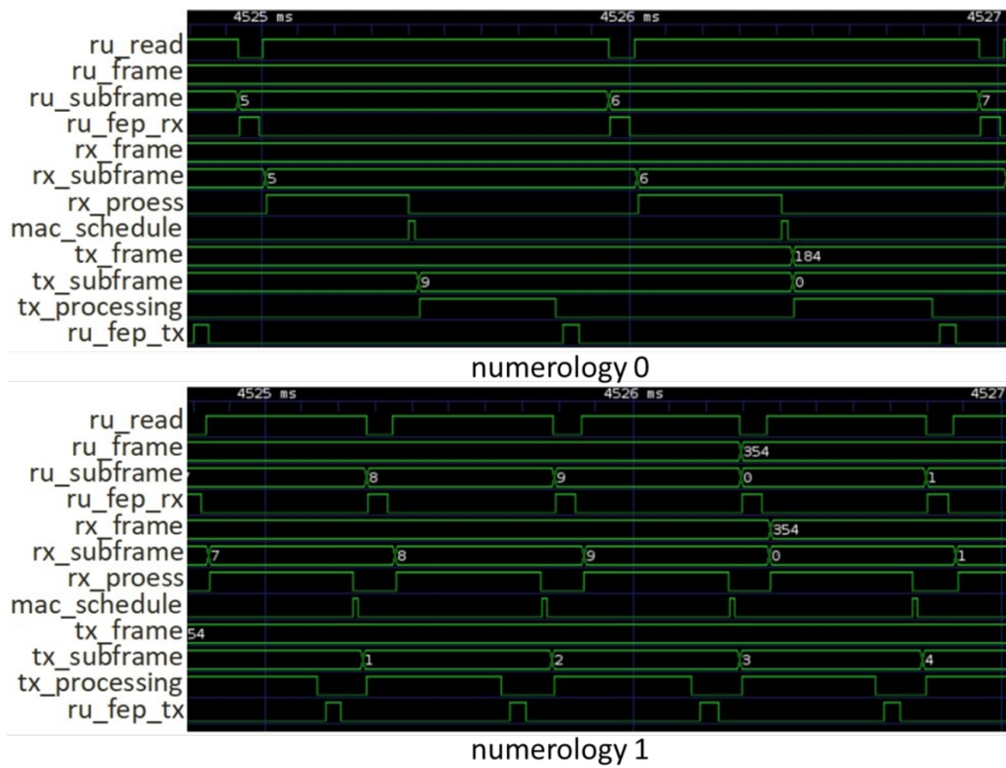

Figure 5.4 – VCD tracking for function and variable

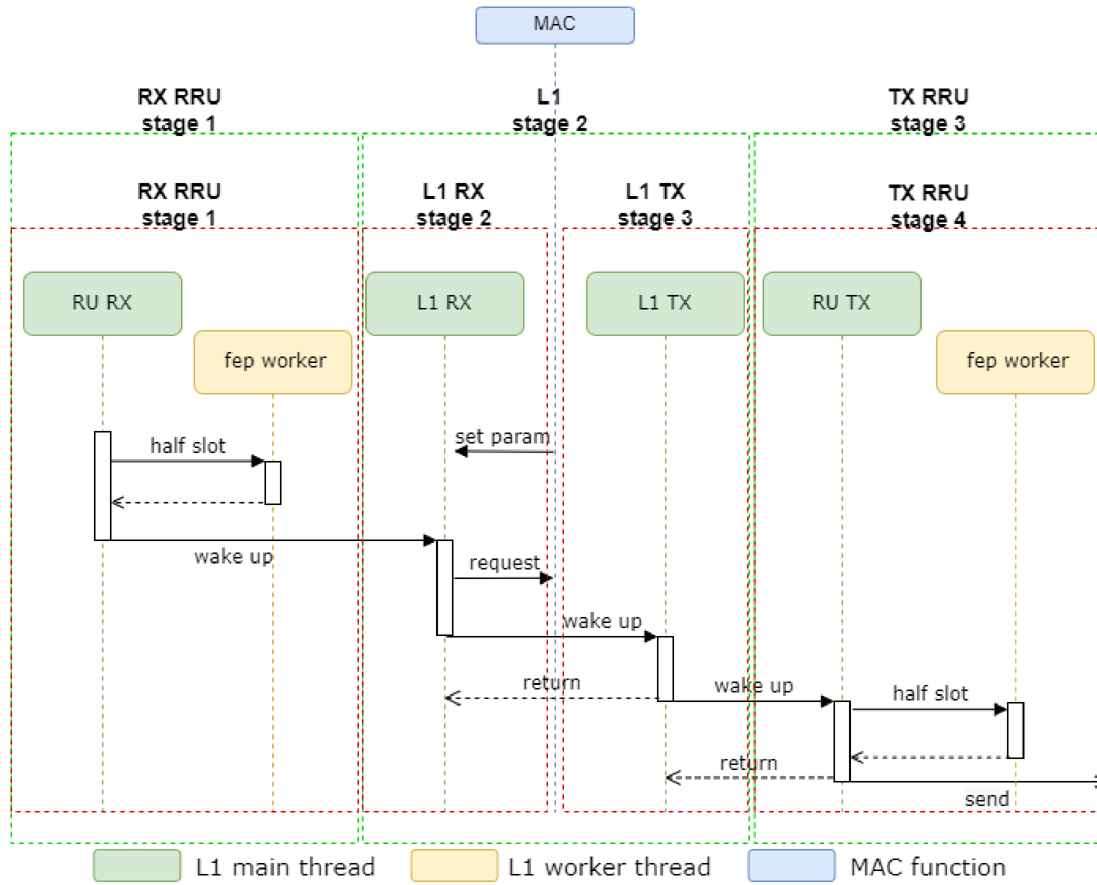

Figure 5.5 – Two different pipeline split

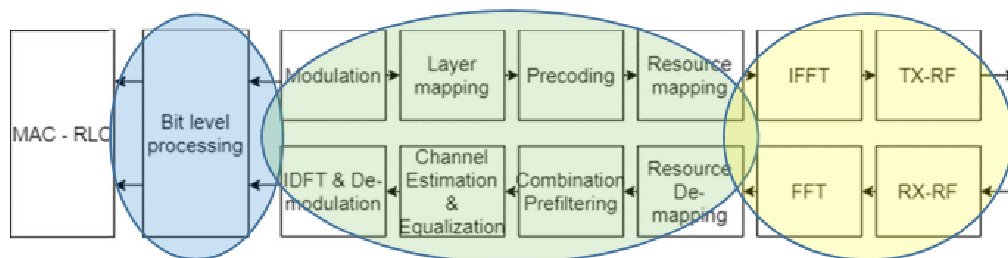

Figure 5.6 – Functions that is categorized in different acceleration group

lack of worker threads to handle the offloading. Since the data will arrive in every time slot but the process won't be able to finish in one, the system won't be able to recover while this is happening. Even if the processing step is complete while the jamming is occurring, the data has already expired. The data was never delivered in time for it to be sent from the fronthaul perspective. One of the testing configurations uses a single worker thread to do the acceleration, dividing the work between coding and decoding. Although this type of structure does aid in the SDR's performance, it is insufficient to maintain the program's stability because the structure was only able to handle the problem when there were few packages. On the other hand, there would be an issue if the structure only had one worker thread to serve one segment. As was previously noted, employing worker threads would add overhead to the execution time, but since coding takes far longer than the overhead, the improvement is still substantial. The overhead will start to affect the execution time if the processing time for coding and decoding is reduced to just one segment.

The thesis makes reference to the late package on a few occasions. Let's have a look at how the SDR with the pipeline topology recovers from late packages. If the TX data didn't reach the fronthaul in time, as indicated by the green slot in figure 5.7, the timestamp will inform the fronthaul that the package exceeded the time restriction and subsequently display the late package signal. The pipeline structure does have the capability to automatically recover from an execution overrun bug, even though having late packages could cause the program to crash. There won't be any jamming at the receiver for the fronthaul because the receiver is not blocked by the other execution. In contrast, if the following slot has less data loading and its processing time is less than two TTI when added to the current slot execution time, the subsequent slot will still have a valid timestamp that arrived in time for the fronthaul, as shown in Figure 5.7. However, if the other process, L1, exceeds the processing time for the TTI, it will result in the late package error also for this slot. A system recovering from a reading error-induced prolonged execution time is shown in Figure 5.7. Even though it is a read-write issue from the fronthaul, it still demonstrates the pipeline structure's capacity to recover from the tardy package.

The outcomes of the channel simulation are shown in Figure 5.9. This simulates the uplink chain, starting with the user equipment (UE) that generates the uplink signal, moving through an Additive White Gaussian Noise (AWGN) channel effect, and finally reaching the OAI base station to decode the signal. At the conclusion, the error rate between the UE-generated signal and the signal being decoded is calculated. It continues to monitor each procedure's timing in the base station in the interim for the timing analysis. The circled result in figure 5.9 corresponds to what is circled in figure 5.6, and it indicates the timing for the uplink procedure with various worker thread counts. The first two results show that cutting the execution time for decoding and other processing in half by increasing the worker thread count from one to two. However, there appears to be performance saturation when comparing the acceleration with either four or eight worker threads. The data amount should be sufficient for the speed to improve by almost half when the number of worker threads is doubled because the setup for the test case was the signal amount that simulated eight antennas. Figure 5.9 illustrates the impact of

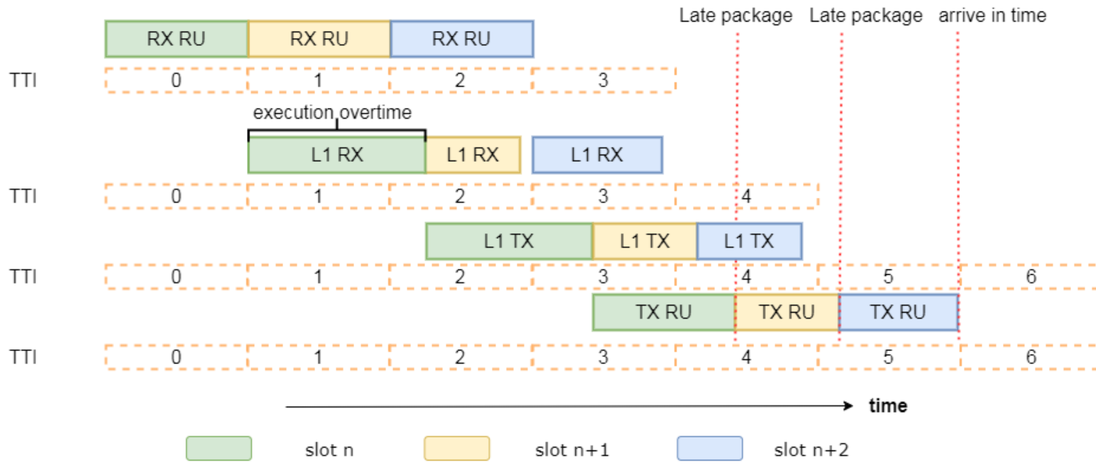

Figure 5.7 – The ability of pipeline structure to recovery from a sudden data rush

Figure 5.8 – The timing log of the recovery of the pipeline structure

the overhead as a result.

Let's move on to the worker acceleration that served at the fronthaul where the test was done in real-time after the simulation results for adding worker threads to offload all the operations in the layer-based procedures. Three situations were tested, each with a different number of antennas, and each scenario was tested with one worker thread per antenna, two worker threads per antenna, and two worker threads per antenna where each worker thread served half of the slot. The USRP n320, which has a maximum of 2 TX and 2 RX antennas, as indicated in table 5.1, and the USRP n310, which has a maximum of 4 TX and 4 RX antennas, as shown in table 5.2, were used in the tests that produced the results that are presented below.

	Tx OFDM execution time		Tx front end process time	
	2x2	1x1	2x2	1x1
Single thread	35.244 us	34.802 us	173.277 us	85.807 us
One thread per antenna	35.955 us		101.138 us	
Two thread per antenna	34.538 us	36.477 us	74.855 us	64.806 us

Table 5.1 – N320 Test result on 60MHz bandwidth

	Tx OFDM execution time		Tx front end process time	
	4x4	2x2	4x4	2x2
Single thread	Late package	34.235 us	Late package	175.157 us
One thread per antenna	45.649 us	33.757 us	134.099 us	100.923 us
Two thread per antenna	X	34.603 us	X	76.411 us

Table 5.2 – N310 Test result on 60MHz bandwidth

It is apparent from the timing results in tables 5.1 and 5.2 that the execution time of an OFDM symbol in a given device appears to vary relatively little under every circumstance, which is to be expected. However, the overall execution time differs dramatically from prior cases where the execution time decreased as the number of worker threads increased. However, the improvement went beyond just dividing the execution time by two. As an example, consider the front-end procedure for one antenna, where overhead is still there as previously described. In addition to the exams, there was a test being run that involved adding threads that were under the control of the main thread. The entire execution time increased despite the fact that the newly added threads did not contain any additional operations. This suggests that using too many threads could make the execution time longer than when using only the right number of threads, in this case two threads per antenna as illustrated in Figure 5.10. Due to an additional attempt to employ the worker thread, which would have resulted in one thread being used for every OFDM symbol, the decision was made to use two threads per antenna. As a result, the data passing time has significantly risen, which ultimately results in an execution time that is practically identical to that of an implementation with a single thread and 1x1 antennas, as illustrated in figure 5.11. Additionally, as the number of antennas required

1 worker thread:

```
gNB RX
Total PHY proc rx                            3145.46 us (100 trials)
 Statistics std=57.59, median=0.00, q1=0.00, q3=0.00 µs (on 0 trials)
|__ RX PUSCH time                           923.92 us (100 trials)
    |__ RX PUSCH channel estimation time    129.48 us (100 trials)
    |__ RX PUSCH Initialization time         49.20 us (100 trials)
    |__ RX PUSCH Symbol Processing time     745.06 us (100 trials)
    |__ ULSCH PTRS Processing time            0.00 us (  0 trials)
|__ ULSCH unscrambling                        0.00 us (  0 trials)
|__ ULSCH total decoding time              2220.70 us (100 trials)
```

2 worker threads:

```
gNB RX
Total PHY proc rx                            1726.40 us (100 trials)
 Statistics std=56.18, median=0.00, q1=0.00, q3=0.00 µs (on 0 trials)
|__ RX PUSCH time                           582.30 us (100 trials)
    |__ RX PUSCH channel estimation time    127.61 us (100 trials)
    |__ RX PUSCH Initialization time         48.97 us (100 trials)
    |__ RX PUSCH Symbol Processing time     405.45 us (100 trials)
    |__ ULSCH PTRS Processing time            0.00 us (  0 trials)
|__ ULSCH unscrambling                        0.00 us (  0 trials)
|__ ULSCH total decoding time              1142.80 us (100 trials)
```

4 worker threads:

```
gNB RX
Total PHY proc rx                            1100.52 us (100 trials)
 Statistics std=56.80, median=0.00, q1=0.00, q3=0.00 µs (on 0 trials)
|__ RX PUSCH time                           420.42 us (100 trials)
    |__ RX PUSCH channel estimation time    129.59 us (100 trials)
    |__ RX PUSCH Initialization time         51.28 us (100 trials)
    |__ RX PUSCH Symbol Processing time     239.31 us (100 trials)
    |__ ULSCH PTRS Processing time            0.00 us (  0 trials)
|__ ULSCH unscrambling                        0.00 us (  0 trials)
|__ ULSCH total decoding time               679.27 us (100 trials)
```

8 worker threads:

```
gNB RX
Total PHY proc rx                             811.22 us (100 trials)
 Statistics std=91.44, median=0.00, q1=0.00, q3=0.00 µs (on 0 trials)
|__ RX PUSCH time                           371.89 us (100 trials)
    |__ RX PUSCH channel estimation time    128.66 us (100 trials)
    |__ RX PUSCH Initialization time         50.29 us (100 trials)
    |__ RX PUSCH Symbol Processing time     192.67 us (100 trials)
    |__ ULSCH PTRS Processing time            0.00 us (  0 trials)
|__ ULSCH unscrambling                        0.00 us (  0 trials)
|__ ULSCH total decoding time               438.51 us (100 trials)
```

Figure 5.9 – Channel simulation result

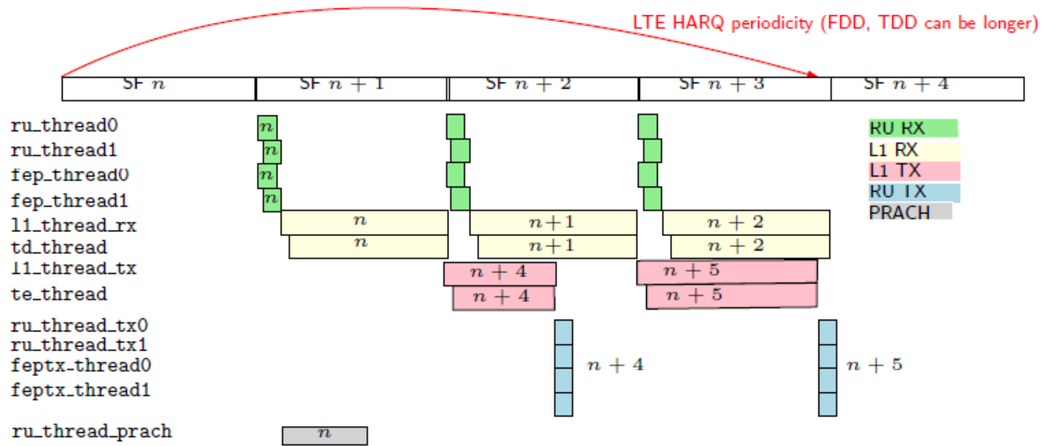

Figure 5.10 – Scenario that have multiple RRU in the pipeline structure

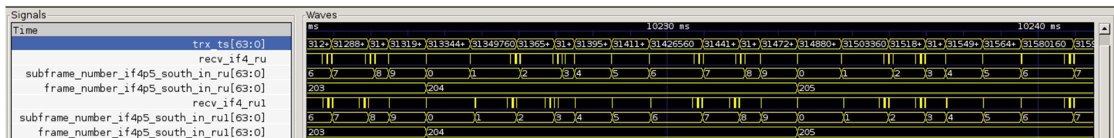

Figure 5.11 – Timing analysis of Transmitting and Receiving every OFDM symbol

rises, there will be an excess of threads running concurrently that cannot be served by the available cores. This will ultimately result in all of the threads still being in a queue while they wait for the cores to process them. An example of executing twelve threads on four physical cores is shown in Figure 5.12. The execution time is almost equal to using four threads when the handover times between each thread are added. In the end, it seems more logical and practical to limit the number of worker threads to one per half of the front-end process.

Figures 5.13 and 5.14 document the outcome of the execution and the connections between the threads and the VCD. On the one-by-one antenna LTE-based SDR, the test was carried out with one thread, which is the main thread only, as shown in Figure 5.13, and two threads, which is with one worker thread for the acceleration to handle the OFDM signals, as shown in Figure 5.14. The timing result for the overall OFDM signal processing, where the yellow circle in Figure 5.6 is, is 60us, which is faster than just one thread process, which is 87us, according to the VCD, which demonstrates that the worker thread was parallel executed from the main thread for the acceleration. The time discrepancy supports the notion that there are thread overhead problems as well. They also fit the results from the 5G SDR tables 5.2 and 5.1.

According to what we previously stated, if the OS distributes threads, it might not be as effective because multiple threads may need to run concurrently on a single

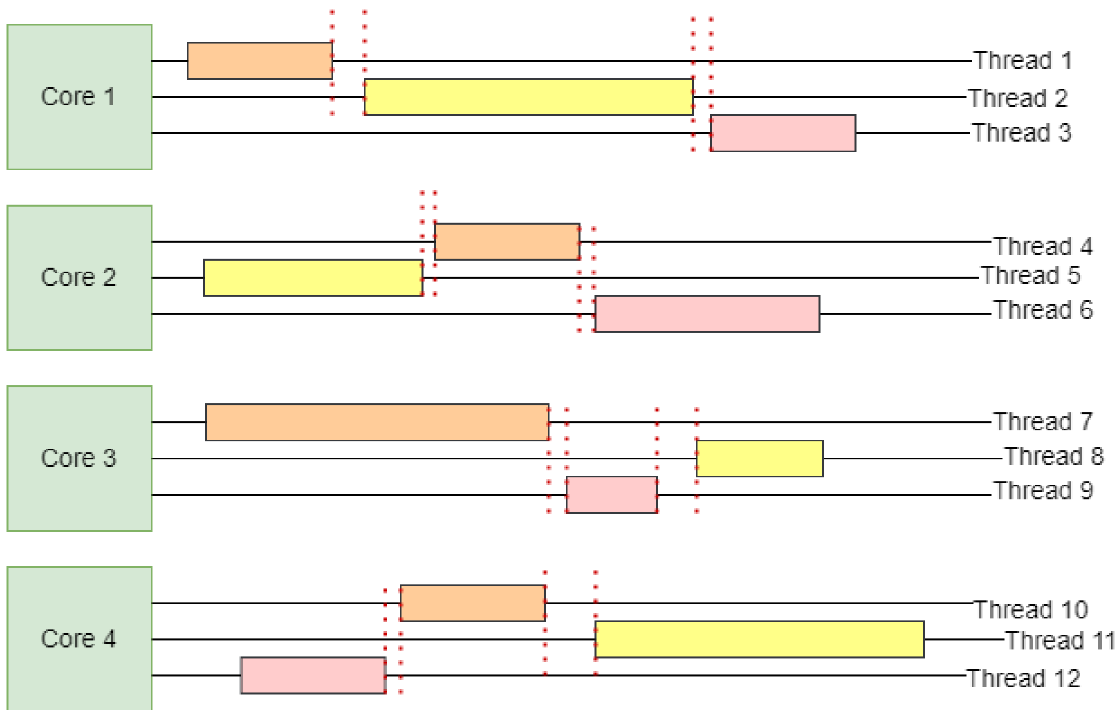

Figure 5.12 – An example of having more threads than cores

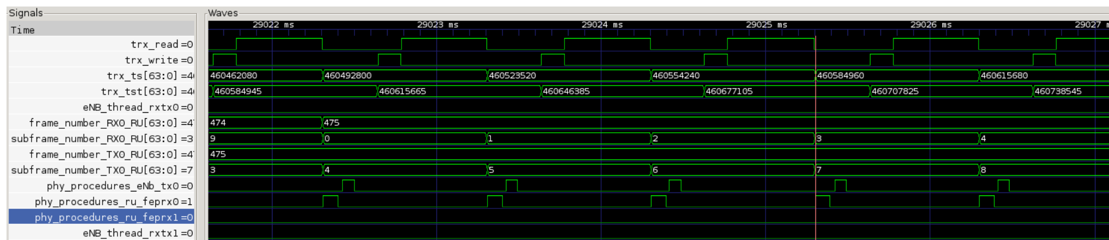

Figure 5.13 – Front end process with single thread

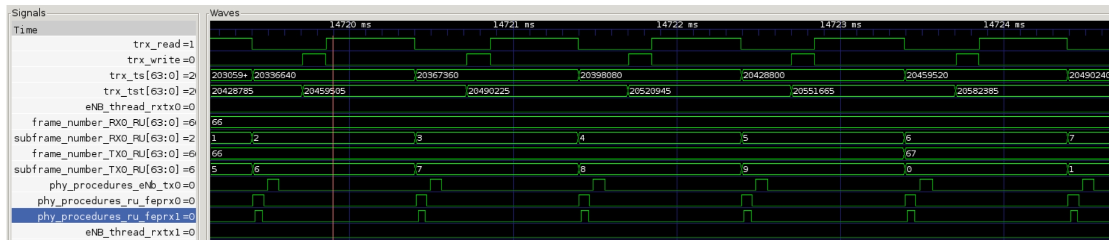

Figure 5.14 – Front end process with two thread

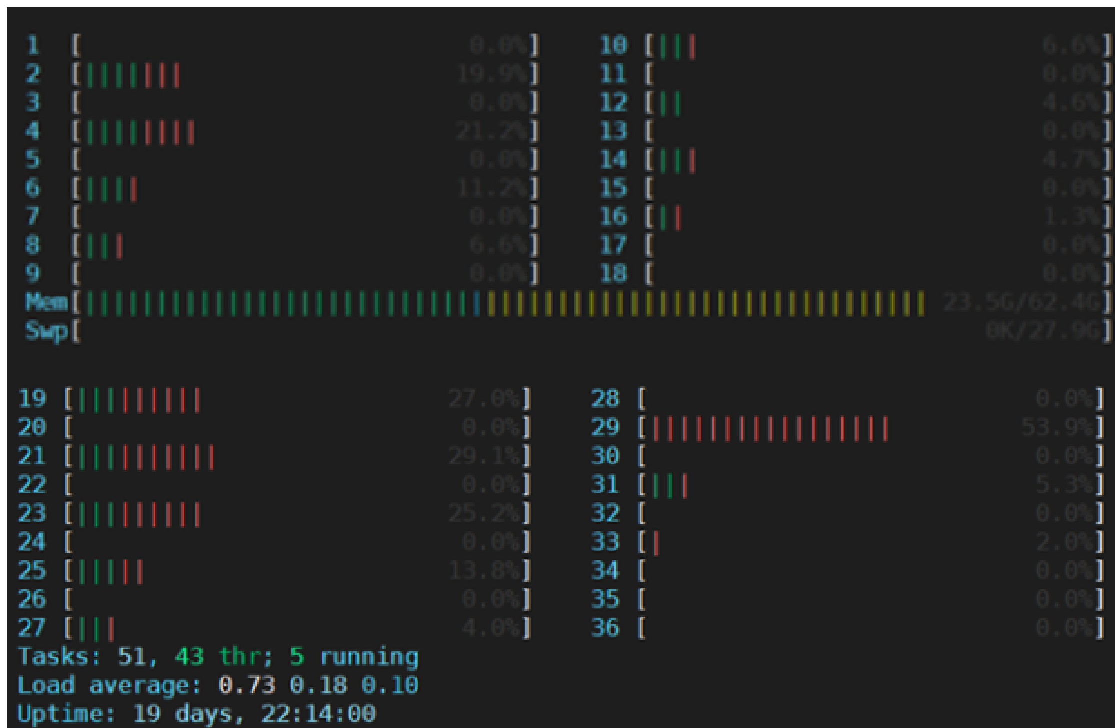

Figure 5.15 – Execution resource distribution

physical thread. Only one task could be carried out at a time because hyperthreading was disabled, and the second work might be delayed, particularly if two threads with significant loads are allocated to the same core. Hardcoding threads to cores during initialization may result in resource waste, but it ensures that no tasks overlap to impair timing performance. Figure 5.15 is a screenshot taken while the OAI SDR is running with a two-by-two antenna and a bandwidth of 60MHz while using allocated threads to cores. It demonstrates that the workload is distributed across multiple cores and that multiple cores are being used simultaneously.

Chapter 6

Conclusion and Future Work

The real-time SDR in the thesis is constructed using OAI. The OAI is accelerated using various techniques. Using threads and recreating the structure are the acceleration techniques. Since the timing limitation in 5G is more stringent, the structure is rebuilt using a pipeline structure in the thesis. This enables to lengthen the process without causing a timing delay issue. Additionally, those heavy-loading procedures are offloaded using a number of worker threads. One of the primary themes discussed in the thesis is how and where to implement worker threads.

The thesis was able to keep the real-time property for the SDR even when integrating OAI to 5G where the timing limitation was much more severe than that in 4G LTE thanks to the introduction of worker threads and pipeline structure. Even with the pipeline layout, there are still problems when the SDR needs to function at maximum capacity and a higher data process is needed. If this occurs, temporal delays will occur because some of the procedures will take too long to complete. Worker threads are available to address this issue. The workload is now multiplied by the number of layers and antennas, rather than being multiplied by the number of antennae, as multi-antennas or even multi-layers would be a feature of 5G. Worker threads are used to parallelize multi-antenna and multi-layer processes since data between various antennae and layers is independent, in addition to offloading the laborious loading procedures.

Regarding upcoming work, as discussed in the previous chapter, thread overhead is a problem even when there are several threads available to offload the workload for accelerated processing. The switching between threads should take less time than it does in the thesis, in theory. The OAI community has already made some progress in partially resolving difficulties with the threading system. However, determining the thread overhead that costs the least amount of timing overhead could aid in improving timing performance, necessitating the use of a new thread handover method.

OS handled the most of the threading organization for the thesis. While creating the threads, a priority parameter can be set. Following that, OS arranged all of the jobs according to the priority parameter. Even if we could instruct the OS to give the desired procedure priority using the priority parameter, all of these tasks would still need to join the task pool with all of the other tasks already running on the processor and compete for access to the CPUs. Tasks for OAI might not be able to obtain the resources to

begin the process immediately away if there are too many high-priority tasks running concurrently. This will influence the timing for OAI and result in the OAI SDR not being stable enough. The goal will be to organize such jobs in a way that is effective while without using up too much processor resources.

Worker thread re-usability is going to be another crucial issue that needs to be taken into consideration since we might have to assign threads to specific CPUs. The thesis uses worker threads that are constructed with a specific topic in mind rather than being general purpose worker threads. Assigning them to CPUs may still result in jamming problems if many executions are required at once because there will be a number of high priority worker threads when the process is needed. The most effective solution, which is also what we want to accomplish, appears to be to have shared worker threads that may handle various tasks and pin them to various CPUs.

The adaptability of SDR has already made it the better option in the current environment. Additionally, its more affordable price contributes to the growth of the marketing. Designing a hardware chip for wireless communication is no longer an option due to the complexity of 5G. However, because to the increased throughput requirements for 5G and future generations, employing GPP-based SDR is currently also experiencing a few challenges. Since an FPGA is programmable and less expensive than creating a hardware chip, using one to support SDR has been proposed as well. While FPGA offers a more promising throughput and speed, SDR built primarily with GPP is easier to maintain, add new features, perform cross-layer cooperation, etc. Due to the advantages that both GPP and FPGA provide, a future for SDR that incorporates both technologies is also possible.

Bibliography

[1] H. ElSawy, H. Dahrouj, T. Y. Al-Naffouri, and M.-S. Alouini, "Virtualized cognitive network architecture for 5g cellular networks," *IEEE Communications Magazine*, vol. 53, no. 7, pp. 78–85, 2015.

[2] W. H. Tuttlebee, "Software-defined radio: facets of a developing technology," *IEEE Personal Communications*, vol. 6, no. 2, pp. 38–44, 1999.

[3] P. Cruz, N. B. Carvalho, and K. A. Remley, "Designing and testing software-defined radios," *IEEE Microwave Magazine*, vol. 11, no. 4, pp. 83–94, 2010.

[4] T. Ulversoy, "Software defined radio: Challenges and opportunities," *IEEE Communications Surveys & Tutorials*, vol. 12, no. 4, pp. 531–550, 2010.

[5] W. H. Tuttlebee, *Software defined radio: enabling technologies.* John Wiley & Sons, 2003.

[6] E. Grayver, *Implementing software defined radio.* Springer Science & Business Media, 2012.

[7] J. R. Machado-Fernández, "Software defined radio: Basic principles and applications," *Revista Facultad de Ingeniería*, vol. 24, no. 38, pp. 79–96, 2015.

[8] M. Dillinger, K. Madani, and N. Alonistioti, *Software defined radio: Architectures, systems and functions.* John Wiley & Sons, 2005.

[9] F. K. Jondral, "Software-defined radio—basics and evolution to cognitive radio," *EURASIP journal on wireless communications and networking*, vol. 2005, no. 3, pp. 1–9, 2005.

[10] C. Zhang, Y.-L. Ueng, C. Studer, and A. Burg, "Artificial intelligence for 5g and beyond 5g: Implementations, algorithms, and optimizations," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 10, no. 2, pp. 149–163, 2020.

[11] M. Taheribakhsh, A. Jafari, M. M. Peiro, and N. Kazemifard, "5g implementation: Major issues and challenges," in *2020 25th International Computer Conference, Computer Society of Iran (CSICC).* IEEE, 2020, pp. 1–5.

[12] R. Hussain, F. Hussain, and S. Zeadally, "Integration of vanet and 5g security: A review of design and implementation issues," *Future Generation Computer Systems*, vol. 101, pp. 843–864, 2019.

[13] R. K. Kodali, G. Swamy, and B. Lakshmi, "An implementation of iot for healthcare," in *2015 IEEE Recent Advances in Intelligent Computational Systems (RAICS)*. IEEE, 2015, pp. 411–416.

[14] K. Cengiz and M. Aydemir, "Next-generation infrastructure and technology issues in 5g systems," *Journal of Communications Software and Systems*, vol. 14, no. 1, pp. 33–39, 2018.

[15] V. W. Wong, R. Schober, D. W. K. Ng, and L.-C. Wang, "Overview of new technologies for 5g systems," *Key Technologies for 5G Wireless Systems*, p. 1, 2017.

[16] M. N. Sadiku and C. M. Akujuobi, "Software-defined radio: a brief overview," *Ieee Potentials*, vol. 23, no. 4, pp. 14–15, 2004.

[17] E. Grayver and A. C. Utter, "Extreme software defined radio – ghz in real time," *2020 IEEE Aerospace Conference*, pp. 1–10, 2020.

[18] G. Sklivanitis, A. Gannon, S. N. Batalama, and D. A. Pados, "Addressing next-generation wireless challenges with commercial software-defined radio platforms," *IEEE Communications Magazine*, vol. 54, no. 1, pp. 59–67, 2016.

[19] L. Karam, I. AlKamal, A. Gatherer, G. A. Frantz, D. V. Anderson, and B. L. Evans, "Trends in multicore dsp platforms," *IEEE signal processing magazine*, vol. 26, no. 6, pp. 38–49, 2009.

[20] M. Palkovic, P. Raghavan, M. Li, A. Dejonghe, L. Van der Perre, and F. Catthoor, "Future software-defined radio platforms and mapping flows," *IEEE Signal Processing Magazine*, vol. 27, no. 2, pp. 22–33, 2010.

[21] Y. Choi, Y. Lin, N. Chong, S. Mahlke, and T. Mudge, "Stream compilation for real-time embedded multicore systems," in *2009 International Symposium on Code Generation and Optimization*. IEEE, 2009, pp. 210–220.

[22] H. Hurskainen, J. Raasakka, T. Ahonen, and J. Nurmi, "Multicore software-defined radio architecture for gnss receiver signal processing," *EURASIP Journal on Embedded Systems*, vol. 2009, pp. 1–10, 2009.

[23] G. J. Minden, J. B. Evans, L. Searl, D. DePardo, V. R. Petty, R. Rajbanshi, T. Newman, Q. Chen, F. Weidling, J. Guffey *et al.*, "Kuar: A flexible software-defined radio development platform," in *2007 2nd IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks*. IEEE, 2007, pp. 428–439.

[24] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "Soda: A high-performance dsp architecture for software-defined radio," *IEEE micro*, vol. 27, no. 1, pp. 114–123, 2007.

[25] T. B. Welch and S. Shearman, "Teaching software defined radio using the usrp and labview," in *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2012, pp. 2789–2792.

[26] A. M. Wyglinski, D. P. Orofino, M. N. Ettus, and T. W. Rondeau, "Revolutionizing software defined radio: case studies in hardware, software, and education," *IEEE Communications magazine*, vol. 54, no. 1, pp. 68–75, 2016.

[27] R.-x. Liu, Z.-g. Zhong, X.-l. Sun *et al.*, "Reverse recognition of ldpc codes based on log-likelihood ratio," in *Journal of Physics: Conference Series*, vol. 1607, no. 1. IOP Publishing, 2020, p. 012106.