Optimization and parallelization methods for software-defined radio

Adrien Cassagne

université de BORDEAUX

Inria

THÈSE PRÉSENTÉE
POUR OBTENIR LE GRADE DE

DOCTEUR DE L'UNIVERSITÉ DE BORDEAUX

École Doctorale Mathématiques et informatique
Spécialité : Informatique

par **Adrien Cassagne**

Méthodes d'optimisation et de parallélisation pour la radio logicielle

Co-directeurs de thèse : Denis Barthou
Christophe Jégo
Co-encadrants de thèse : Olivier Aumage
Camille Leroux

préparée au Centre de Recherche Inria Bordeaux - Sud-Ouest et au Laboratoire de l'Intégration du Matériau au Système (IMS)

soutenue le 8 décembre 2020

Jury :

Lionel Lacassagne	-	Professeur des Universités	-	Sorbonne Université	*Rapporteur*
Matthieu Gautier	-	Maître de Conférences, HDR	-	Université de Rennes 1	*Rapporteur*
Michel Jezequel	-	Professeur des Universités	-	IMT Atlantique	*Président*
Cédric Bastoul	-	Directeur Scientifique, HDR	-	Huawei Paris	*Examinateur*
Camille Leroux	-	Maître de Conférences	-	Bordeaux INP	*Examinateur*
Olivier Aumage	-	Chargé de Recherche, HDR	-	Inria	*Examinateur*
Christophe Jégo	-	Professeur des Universités	-	Bordeaux INP	*Co-directeur*
Denis Barthou	-	Professeur des Universités	-	Bordeaux INP	*Co-directeur*

Thèse réalisée au Centre de Recherche Inria Bordeaux - Sud-Ouest,
au sein de l'équipe projet STORM.

Université de Bordeaux
Centre de Recherche Inria Bordeaux - Sud-Ouest
200 Avenue de la Vieille Tour
33405 Talence

Thèse réalisée au Laboratoire de l'Intégration du Matériau au Système (IMS)
de Bordeaux, au sein de l'équipe CSN du groupe Conception.

Université de Bordeaux, Laboratoire IMS
UMR 5218 CNRS - Bordeaux INP
351 Cours de la Libération
Bâtiment A31
33405 Talence Cedex

Remerciements

Je tiens en premier lieu à remercier les membres du jury pour l'intérêt qu'ils ont porté à ces travaux. Je remercie donc chaleureusement Monsieur Lionel Lacassagne, professeur de l'Université de Sorbonne et Monsieur Matthieu Gautier, maître de conférences de l'Université de Rennes de m'avoir fait l'honneur de rapporter ce travail de thèse. Je remercie aussi Monsieur Michel Jezequel, professeur de l'Institut Mines-Télecom Atlantique d'avoir présidé le jury ainsi que Monsieur Cédric Bastoul, directeur scientifique chez Huawei, d'avoir accepté d'examiner ma thèse.

Je remercie sincèrement mes encadrants qui ont été exceptionnels, chacun sur des aspects différents. Camille, merci pour tes explications claires et imagées, pour ta croyance sans faille dans le projet, et bien sûr pour les moments passés en dehors. Olivier, merci pour ton expertise précieuse, ton soutient lors des changements de direction et pour ta patience et ton aide lors des phases de rédaction. Christophe, merci de m'avoir toujours soutenu, en commençant par le réagencement de notre espace de travail jusqu'au soutient infaillible dans le projet. Dans ces conditions il était naturel de pouvoir évoluer sereinement. Denis, merci de m'avoir fait confiance et de m'avoir donné l'opportunité de pouvoir intégrer le monde de la recherche. Merci d'avoir su me guider dans ce projet, ton recul et tes conseils avisés m'ont beaucoup apporté.

Merci à tous mes collègues de l'équipe CSN d'avoir contribué à l'excellente ambiance de travail. Je pense à Baptiste, Bertrand, Camilo, Dominique, Guillaume B., Guillaume D., Imen, Jérémie, Jonathan, Logan, Malek, Mathieu, Olivier, Thibaud, Vincent, Yann et Yassine.

Et, pour les mêmes raisons, aussi un grand merci à tous mes collègues de l'équipe STORM. Je pense à Alexis, Amina, Baptiste, Célia, Chiheb, Corentin, Emmanuelle, Hugo, Jean-Marie, Kun, Léo, Maël, Marie-Christine, Mariem, Mehdi, Nathalie, PAW, Philippe, Pierre, Raymond, Romain, Sabrina, Samuel et Yanis.

Je tiens à remercier tout particulièrement le noyau dur de l'équipe AFF3CT avec qui nous avons partagé des moments forts. Romain, merci pour tes nombreuses explications, ta motivation sans faille et tes qualités humaines, ça a été un réel plaisir de pouvoir travailler avec toi. Olivier, merci d'avoir autant accroché au projet, d'avoir toujours su proposer des nouvelles idées et de m'avoir supporté. Tu es incontestablement un des plus grand contributeur et une personne avec qui j'ai adoré travailler. Mathieu, que dire, merci d'avoir toujours cru au projet, de l'avoir adopté et d'avoir été le premier doctorant à prouver qu'il était possible d'en tirer parti et de l'enrichir. Je suis aussi extrêmement fier du travail accompli ensemble.

Pour terminer, je veux dire un mot à tous les gens qui me sont chers en dehors du travail et qui se reconnaîtront. Vous avez joué un rôle déterminant dans le bon déroulement de cette thèse, à différents moments et à différents niveaux. Pour cela je vous remercie infiniment et j'espère avoir la chance de vous avoir à mes côtés encore longtemps.

Résumé

Une radio logicielle (en anglais *Software-Defined Radio* ou SDR) est un système de communications numériques reconfigurable utilisant des techniques de traitement du signal sur des architectures numériques programmables. Avec l'émergence de nouveaux standards de communications complexes et la puissance de calcul grandissante des processeurs généralistes, il devient intéressant d'échanger l'efficacité énergétique des architectures dédiées contre la souplesse et la facilité d'implémentation sur processeurs généralistes.

Même lorsque l'implémentation d'un traitement numérique est finalement faite sur une puce dédiée, une version logicielle de ce traitement s'avère nécessaire en amont pour s'assurer des bonnes propriétés de la fonctionnalité. Cela est généralement réalisé via la simulation. Les simulations sont cependant souvent coûteuses en temps de calcul. Il n'est pas rare de devoir attendre plusieurs jours voire plusieurs semaines pour évaluer les performances du modèle fonctionnel d'un système.

Dans ce contexte, cette thèse propose d'étudier les algorithmes les plus coûteux en temps de calcul dans les chaînes de communication numériques actuelles. Ces algorithmes sont le plus souvent présents dans des décodeurs de codes correcteurs d'erreurs au niveau récepteur. Le rôle du codage canal est d'accroître la robustesse vis à vis des erreurs qui peuvent apparaître lorsque l'information transite au travers d'un canal de transmission. Trois grandes familles de codes correcteurs d'erreurs sont étudiées dans nos travaux, à savoir les codes LDPC, les codes polaires et les turbo codes. Ces trois familles de codes sont présentes dans la plupart des standards de communication actuels comme le Wi-Fi, l'Ethernet, les réseaux mobiles 3G, 4G et 5G, la télévision numérique, etc. Les décodeurs qui en découlent proposent le meilleur compromis entre la résistance aux erreurs et la vitesse de décodage. Chacune de ces familles repose sur des algorithmes de décodage spécifiques. Un des enjeux principal de cette thèse est de proposer des implémentations logicielles optimisées pour chacune des trois familles. Des réponses sont apportées de façon spécifique puis des stratégies d'optimisation plus générales sont discutées. L'idée est d'abstraire des stratégies d'optimisation possibles en étudiant un sous-ensemble représentatif de décodeurs.

Enfin, la dernière partie de cette thèse propose la mise en œuvre d'un système de communications numériques complet à l'aide de la radio logicielle. En s'appuyant sur les implémentations rapides de décodeurs proposées, un émetteur et un récepteur compatibles avec le standard DVB-S2 sont implémentés. Ce standard est typiquement utilisé pour la diffusion de contenu multimédia par satellite. À cette occasion, un langage dédié à la radio logicielle est développé pour tirer parti de l'architecture parallèle des processeurs généralistes actuels. Le système atteint des débits suffisants pour être déployé en condition opérationnelle.

Les différentes contributions des travaux de thèse ont été faites dans une dynamique d'ouverture, de partage et de réutilisabilité. Il en résulte une bibliothèque à code source ouvert nommée AFF3CT pour *A Fast Forward Error Correction Toolbox*. Ainsi, tous les résultats proposés dans cette thèse peuvent aisément être reproduits et étendus. Cette philosophie est détaillée dans un chapitre spécifique du manuscrit de thèse.

Mots clefs : Radio logicielle, Simulation fonctionnelle, Codes correcteurs d'erreurs, Implémentation logicielle, Optimisation, Parallélisation, Code source ouvert

Abstract

A software-defined radio is a radio communication system where components traditionally implemented in hardware are instead implemented by means of software. With the growing number of complex digital communication standards and the general purpose processors increasing power, it becomes interesting to trade the energy efficiency of the dedicated architectures for the flexibility and the reduced time to market on general purpose processors.

Even if the resulting implementation of a signal processing is made on an application-specific integrated circuit, the software version of this processing is necessary to evaluate and verify the correct properties of the functionality. This is generally the role of the simulation. Simulations are often expensive in terms of computational time. To evaluate the global performance of a communication system can require from few days to few weeks.

In this context, this thesis proposes to study the most time consuming algorithms in today's digital communication chains. These algorithms often are the channel decoders located on the receivers. The role of the channel coding is to improve the error resilience of the system. Indeed, errors can occur at the channel level during the transmission between the transmitter and the receiver. Three main channel coding families are then presented: the LDPC codes, the polar codes and the turbo codes. These three code families are used in most of the current digital communication standards like the Wi-Fi, the Ethernet, the 3G, 4G and 5G mobile networks, the digital television, etc. The resulting decoders offer the best compromise between error resistance and decoding speed known to date. Each of these families comes with specific decoding algorithms. One of the main challenge of this thesis is to propose optimized software implementations for each of them. Specific efficient implementations are proposed as well as more general optimization strategies. The idea is to extract the generic optimization strategies from a representative subset of decoders.

The last part of the thesis focuses on the implementation of a complete digital communication system in software. Thanks to the efficient decoding implementations proposed before, a full transceiver, compatible with the DVB-S2 standard, is implemented. This standard is typically used for broadcasting multimedia contents via satellite. To this purpose, an embedded domain specific language targeting the software-defined radio is introduced. The main objective of this language is to take advantage of the parallel architecture of the current general purpose processors. The results show that the system achieves sufficient throughputs to be deployed in real-world conditions.

These contributions have been made in a dynamic of openness, sharing and reusability, it results in an open source library named AFF3CT for A Fast Forward Error Correction Toolbox. Thus, all the results proposed in this thesis can easily be reproduced and extended. This philosophy is detailed in a specific chapter of the thesis manuscript.

Keywords: Software-Defined Radio, Functional Simulation, Error Correcting Codes, Software Implementation, Optimization, Parallelization, Open Source Code

Résumé étendu

Chapitre 1 - Contexte et objectifs

Organisation

Ce chapitre présente le contexte des systèmes de communications numériques. Il a pour but de définir les notions qui seront réutilisées dans le manuscrit et de donner une vue globale. Il définit aussi les principaux objectifs de cette thèse. La première partie présente le principe des systèmes de communications numériques avec ses différentes composantes : l'émetteur, le canal et le récepteur. Les métriques les plus couramment utilisées dans les communications numériques sont présentées avec notamment la définition du taux d'erreur binaire (*Bit Error Rate*, BER) et du taux d'erreur trame (*Frame Error Rate*, FER). La deuxième partie détaille le modèle de canal qui correspond à un ajout d'un bruit blanc gaussien (*Additive White Gaussian Noise*, AWGN) et la modulation numérique binaire par changement de phase (*Binary Phase-Shift Keying*, BPSK) utilisés tout au long du manuscrit. Une caractérisation du rapport signal sur bruit (*Signal-to-Noise Ratio*, SNR) est donnée ainsi que la notion de probabilité à la sortie du canal et du démodulateur. La troisième partie présente les familles de code correcteur d'erreurs considérées dans ce manuscrit, à savoir les codes LDPC, les codes polaires et les turbo-codes. Les traitements de codage (situé au niveau de l'émetteur) et de décodage (situé au niveau du récepteur) correspondants sont détaillés pour chaque famille de code. Ces familles de code sont utilisées dans la plupart des standards de télécommunication actuels et engendrent une complexité calculatoire élevée. Par conséquent, ce sont de bons candidats d'étude. Dans la dernière partie, deux contextes applicatifs majeurs sont détaillés pour les familles de code considérées. La "simulation fonctionnelle" permet la conception et la validation d'un schéma de codage. La "radio logicielle" (*Software-Defined Radio*, SDR) est un système de communication radio où tous les composants sont implémentés avec des blocs logiciels (par opposition à des implémentations matérielles plus couramment utilisées dans ce domaine).

Objectifs de la thèse

À l'aube de la cinquième génération des standards pour la téléphonie mobile (5G), le défi consiste maintenant à concevoir des systèmes de communication capables de transmettre une grande quantité de données en peu de temps, à un faible coût énergétique et dans des environnements très variés. Les chercheurs s'efforcent d'affiner encore les schémas de codage existants, afin d'obtenir de faibles taux d'erreur résiduels grâce à des processus de décodage rapides, souples et aussi peu complexe que possible.

Simulation fonctionnelle La validation d'un système de codage nécessite d'estimer son taux d'erreur. En général, il n'existe pas de modèle mathématique simple pour décrire ce type de performance. La seule solution pratique consiste à effectuer une simulation de type Monte Carlo de l'ensemble de la chaîne de transmission. Cela signifie que certaines données sont générées,

encodées, modulées, bruitées, décodées de manière aléatoire, et que les performances sont ensuite estimées en mesurant le taux d'erreur binaire (BER) et le taux d'erreur trame (FER) à la fin de la chaîne de communication (après décision du décodeur). Ce processus a l'avantage d'être universel mais il entraîne également trois problèmes principaux :

1. **Temps de simulation :** ~ 100 trames erronées doivent être simulées pour estimer avec précision les BER/FER. Ainsi, la mesure d'un FER de 10^{-7} nécessite la simulation de la transmission de $\sim 100 \times 10^7 = 10^9$ trames. En supposant une trame de 1000 bits, le simulateur doit alors traiter la transmission de 10^{12} bits. En gardant à l'esprit que la complexité de calcul de l'algorithme de décodage peut être importante, plusieurs semaines voire plusieurs mois peuvent être nécessaires pour estimer avec précision les BER/FER d'un schéma de codage (en particulier si le taux d'erreur est faible).
2. **Hétérogénéité algorithmique :** un grand nombre de codes correcteurs d'erreurs ont été conçus au fil des années. Pour chaque famille de code, plusieurs configurations de décodage sont possibles. Si il est simple de décrire un schéma de codage unique, il est plus difficile d'avoir une description logicielle unifiée qui prenne en charge tous les schémas de codage et les algorithmes de décodage associés. Cette difficulté provient de l'hétérogénéité des structures de données nécessaires pour décrire les différents schémas de codage canal : les turbo-codes sont basés sur des treillis, les codes LDPC sont bien définis par des graphes bipartis et les codes polaires sont décodés efficacement à l'aide d'arbres binaires.
3. **Reproductibilité :** il est généralement fastidieux de reproduire des résultats issus de la littérature. Cela peut s'expliquer par la grande quantité de paramètres empiriques nécessaires pour définir un système de communication, et par le fait que tous ne sont pas toujours rapportés dans les publications. En outre, le code source des simulateurs est rarement accessible au public. Par conséquent, beaucoup de temps est passé à "réinventer la roue" juste pour pouvoir comparer de nouveaux résultats avec l'état de l'art.

Radio logicielle Le paradigme de la radio logicielle (*Software-Defined Radio*, SDR) est désormais considéré pour des systèmes de communications numériques réels. Pour répondre aux contraintes posées par les systèmes temps réel, voici les principaux défis à relever :

1. **Haut débit :** les nouvelles applications comme le *streaming* vidéo, peuvent être très gourmandes en données. Par conséquent, les tâches de calcul intensif au sein de l'émetteur et du récepteur doivent être bien optimisées pour atteindre des niveaux de performance comparables à des implémentation matérielles.
2. **Faible latence :** atteindre un débit élevé n'est pas toujours la contrainte majeure, par exemple, dans les applications d'audio-conférence, il est inconfortable de percevoir un retard lorsque les gens parlent.
3. **Flexibilité :** les implémentations logicielles doivent pouvoir s'adapter à diverses configurations. Par exemple, lorsque le rapport signal sur bruit (SNR) change, le décodeur doit être capable se s'adapter "à la volée" à de nouveaux taux de codage.
4. **Portabilité :** les solutions proposées peuvent être déployées sur des serveurs haut de gamme ainsi que dans des systèmes embarqués à faible consommation énergétique. De plus, de nombreux systèmes d'exploitation coexistent et il est important de pouvoir supporter les plus communs comme Windows, macOS et Linux.

Chapitre 2 - Stratégies d'optimisation

Organisation

Ce chapitre se concentre sur les stratégies d'optimisation dédiées aux algorithmes de communications numériques. Nos contributions sont divisées en deux parties : 1) les stratégies génériques et 2) les optimisations spécifiques. La première partie décrit les stratégies génériques que nous avons proposées pour optimiser les algorithmes présents dans les récepteurs de systèmes de communications numériques. Il s'avère que la vectorisation est une des clefs pour implémenter des solutions logicielles efficaces. Une bibliothèque dédiée ainsi que des niveaux de parallélisme génériques sont proposés. La seconde partie est consacrée à l'implémentation logicielle efficace d'un sous-ensemble représentatif de décodeurs pour les trois grandes familles de code abordées plus tôt, à savoir : les codes LDPC, les codes polaires et les turbo-codes.

Principaux résultats

En premier lieu, des stratégies génériques pour l'implémentation efficace d'algorithmes sur processeurs généralistes (CPUs) sont présentées. Une contribution majeure dans ce chapitre est la proposition de MIPP : une bibliothèque qui encapsule les instructions vectorielles. L'idée est d'abstraire les types de données et les multiples jeux d'instructions vectoriels existants afin de proposer des implémentations logicielles "universelles" et efficaces des algorithmes présents dans les récepteurs de systèmes de communications numériques. Nous montrons que MIPP n'introduit presque pas de surcoût par rapport aux fonctions intrinsèques (ou du code assembleur). MIPP fonctionne aussi bien sur des représentations en virgule flottante et que sur des représentations en virgule fixe. Pour les algorithmes présents dans les récepteurs de systèmes de communications numériques, les représentations en virgule fixe sont très intéressantes car elles permettent de traiter un plus grand nombre d'éléments dans les registres vectoriels, avec un impact modéré sur les performances de décodage. Pour résumer, **MIPP améliore la flexibilité et la portabilité du code source tout en conservant un même niveau de performance.** Notez que la bibliothèque MIPP a été valorisée suite à une publication dans une conférence scientifique internationale [7].

Dans une deuxième partie, deux grandes stratégies de vectorisation sont explicitement définies et présentées. La stratégie intra-trame fonctionne sur une seule trame en s'appuyant sur le parallélisme inhérent à l'algorithme. La stratégie inter-trames fonctionne quant à elle sur plusieurs trames en même temps. La stratégie intra-trame peut à la fois augmenter le débit et diminuer la latence. Au contraire, la stratégie inter-trames n'améliore pas la latence. Elle s'accompagne cependant d'une efficacité vectorielle potentiellement plus élevée et peut conduire à des débits très élevés. Ces deux stratégies peuvent être appliquées à tous les blocs de traitement des chaînes de communications numériques. **Les stratégies intra-trame et inter-trames constituent donc un point clé pour résoudre le problème de l'hétérogénéité algorithmique.**

Les dernières parties du chapitre se concentrent sur la conception d'implémentations logicielles efficaces des algorithmes de décodage présentés dans le chapitre précédent. Les décodeurs LDPC, les décodeurs polaires et le turbo-décodeur sont compatibles avec la stratégie inter-trames, tandis que les décodeurs polaires sont aussi compatibles avec la stratégie intra-trame. En fonction des familles de codes, nous nous concentrons sur différentes contraintes. **Les décodeurs LDPC ont été mis en œuvre pour prendre en charge de nombreuses variantes et donc pour maximiser la flexibilité** au prix de débits plus faibles et de latences plus élevées par rapport à d'autres

travaux. Ce choix permet d'évaluer les performances de décodage de nombreuses combinaisons algorithmiques. **Dans les décodeurs polaires, la flexibilité ainsi que des optimisations agressives sont considérées et comparées. Ces dernières permettent d'atteindre de très faibles latences. Enfin, le turbo-décodeur se concentre sur l'obtention de débits les plus élevés possibles.** Certaines spécialisations sont faites pour le standard LTE. Il est à noter que la plupart des implémentations logicielles proposées ont fait l'objet de publications dans des conférences et des revues scientifiques internationales [2, 3, 4, 5, 6].

Chapitre 3 - AFF3CT : une boîte à outils pour le codage canal

Organisation

Ce chapitre est consacré à la présentation de notre boîte à outils *open-source* nommée AFF3CT. La première partie décrit les principaux prérequis en fonction de quatre objectifs : l'implémentation d'un logiciel hautement performant, la prise en charge de l'hétérogénéité algorithmique, la portabilité et la reproductibilité. Dans la deuxième partie, AFF3CT est comparé aux autres bibliothèques logicielles de codage canal `C/C++` existantes. La troisième partie présente AFF3CT comme une bibliothèque dédiée aux algorithmes de communications numériques. L'architecture et les fonctionnalités du logiciel sont décrites. Ensuite, des exemples d'utilisation de la bibliothèque sont donnés en `C++` et en MATLAB®. La quatrième partie se concentre sur le simulateur AFF3CT qui est livré dans la boîte à outils. Un aperçu des explorations possibles est donné et notre comparateur de BER/FER est présenté. À la fin, la stratégie de test d'AFF3CT est expliquée. Une dernière partie est consacrée à l'impact d'AFF3CT dans les contextes industriels et universitaires. Une revue des publications scientifiques qui ont utilisé AFF3CT est donnée.

Principaux résultats

Tout d'abord, l'accent est mis sur **la bibliothèque AFF3CT qui vient avec une architecture logicielle qui permet l'hétérogénéité algorithmique**. De nombreuses familles de code correcteur d'erreurs sont supportées comme les codes LDPC, les codes polaires, les turbo-codes, les turbo-codes produit, les codes convolutifs, les codes BCH, les codes Reed-Solomon, etc. **À notre connaissance, AFF3CT est la bibliothèque qui offre le support le plus complet pour les algorithmes associés au codage canal.** AFF3CT est également livré avec de multiples modèles de canaux (AWGN, Rayleigh, BEC, BSC, etc.) et différents schémas de modulation numérique (PSK, QAM, PAM, OOK, CPM, SCMA, etc.). Toutes ces implémentations logicielles efficaces d'algorithmes peuvent être utilisées à partir d'interfaces. Des exemples d'utilisation de la bibliothèque sont donnés en `C++` natif ou en utilisant l'encapsulation MATLAB®. La boîte à outils AFF3CT a fait l'objet de publications dans une conférence et une revue scientifique internationale [8, 1].

AFF3CT est également fourni avec un simulateur fonctionnel de BER/FER. Toutes les caractéristiques précédemment énumérées peuvent être simulées sur différents paramètres. **Sa capacité à explorer une grande variété de paramètres est démontrée.** De nombreux paramètres peuvent être modifiés comme le nombre d'itérations de décodage, les approximations dans l'implémentation des algorithmes, la quantification des données dans les décodeurs, etc. Certains de ces paramètres sont présentés pour les décodeurs introduits dans les chapitres précédents. Ce sujet a été valorisé par un article dans une conférence nationale [9].

AFF3CT est conçu pour permettre la reproductibilité des résultats scientifiques. Un outil de comparaison des performances de décodage (BER/FER) a été ajouté pour permettre une recherche facile dans une base de données d'environ 500 références pré-simulées. Toutes ces références sont des résultats simulés avec AFF3CT qui peuvent être aisément reproduits. À cette fin, une suite de tests a été mis en place. Chaque fois qu'il y a une modification du code source, la base de données des références est rejouée pour éviter des problèmes de régression. Ces tests sont également effectués sur plusieurs architectures (x86 et ARM®) et systèmes d'exploitation (Windows, macOS et Linux) afin de garantir que la portabilité soit toujours conservée.

La dernière partie du chapitre traite de l'impact d'AFF3CT dans la communauté. **Il est montré que de plus en plus d'utilisateurs adoptent la boîte à outils AFF3CT aussi bien dans l'industrie que dans les milieux académiques.** Les contextes applicatifs sont variés et vont de la validation des performances de décodage à l'utilisation de sous-parties spécifiques de la bibliothèque. Les contributions externes sont cependant encore rares.

Chapitre 4 - Évaluation et comparaison des performances

Organisation

Ce chapitre propose d'évaluer les différentes contributions exposées dans les chapitres précédents. Les trois premières parties se concentrent sur les implémentations logicielles efficaces de décodeurs LDPC, de décodeurs polaires et d'un turbo-décodeur. Le débit, la latence et l'efficacité énergétique sont étudiés et comparés avec d'autres travaux de la littérature. La quatrième partie résume les implémentations de décodeurs logiciels les plus efficaces que nous avons trouvées dans la littérature. Cet état de l'art est décomposé en trois catégories distinctes : une pour les décodeurs LDPC, une pour les décodeurs polaires et une pour les turbo-décodeurs. Certaines métriques sont spécifiées afin de faciliter la comparaison entre les différentes publications. La dernière partie est consacrée à une analyse des performances du simulateur AFF3CT. Une chaîne de communications numériques représentative est définie et évaluée en séquentiel et en parallèle. Cette chaîne utilise un décodeur polaire rapide évalué plus tôt dans le chapitre.

Principaux résultats

Pour les décodeurs LDPC et les turbo-décodeurs, la stratégie inter-trames a été appliquée. Elle permet d'obtenir des débits comparables aux meilleurs travaux de la littérature. Toutefois, les latences ne sont pas compétitives avec les meilleures implémentations de type intra-trame que l'on trouve dans la littérature. L'implémentation inter-trames proposée est alors davantage orientée vers la simulation ou vers des applications en temps réel qui ne nécessitent pas une faible latence comme le *streaming* vidéo, par exemple. Pour les décodeurs polaires, les stratégies inter-trames et intra-trame ont été implémentées. Il en résulte un *framework* complet qui peut s'adapter à de nombreux contextes applicatifs. **Les décodeurs proposés sont parmi les plus rapides de la littérature. Ils peuvent également être très flexibles avec les implémentations dynamiques ou spécialisés pour des performances maximales avec la technique de génération de code source.** Pour tous les décodeurs proposés (code LDPC, code polaire et turbo-code), le niveau de généricité est l'une de nos principales contributions. **Les implémentations sont capables de s'adapter à différentes architectures de CPU ainsi que de supporter de nombreuses variantes algorithmiques.** De plus, chacune des implémentations présentées est capable de travailler à un niveau proche des performances de décodage de référence. La

plupart des résultats obtenus ont été publiés dans des conférences et des revues scientifiques internationales [2, 3, 4, 5, 6].

Les "Temples de la renommée" (*Hall of Fames*, HoFs) des décodeurs logiciels sont ensuite présentés. Ces HoFs représentent des états de l'art complets de chaque famille de code correcteur d'erreurs. Les implémentations de décodeur proposées dans la thèse sont comparées avec les autres travaux de la littérature. Ces HoFs permettent de comparer les implémentations CPU et GPU. Certaines mesures telles que le débit normalisé, le TNDC et la consommation d'énergie sont définies. Les résultats montrent que ces dernières années, les implémentations CPU sont plus efficaces que les implémentations GPU en termes de débit, de latence et d'efficacité énergétique. L'un des principaux problèmes des implémentations basées sur GPU est le temps de transfert nécessaire entre le CPU et le GPU. Un autre problème majeur vient de l'architecture intrinsèque des GPUs qui nécessite un parallélisme très élevé pour être efficace. Il n'est pas toujours possible de tirer parti de ce niveau élevé de parallélisme dans les algorithmes de décodage de code correcteur d'erreurs. **Par conséquent, en général, les CPUs sont plus adaptés pour des implémentations logicielles à faible latence que les GPUs.**

La dernière partie de ce chapitre est consacrée aux performances du simulateur AFF3CT. Une chaîne de communications numériques entièrement vectorisée est proposée pour l'évaluation. Des performances sur un seul cœur de calcul CPU sont d'abord présentées. Il en résulte qu'AFF3CT fonctionne le plus rapidement sur les derniers processeurs Intel® Gold qui supportent le jeu d'instructions vectorielles "AVX-512". Ensuite, les performances sur plusieurs cœurs de calcul sont mises à l'épreuve. Dans ce cadre, les processeurs AMD® EPYC s'avèrent être les plus performants : le débit utile de la chaîne atteint 11 Gb/s. Même si les processeurs AMD® EPYC ne prennent en charge que les instructions de type "AVX", il semble que l'architecture Zen 2 soit bien équilibrée entre la puissance de calcul et la vitesse de la mémoire. Enfin, la capacité multi-nœuds du simulateur AFF3CT est testée et une accélération linéaire est observée sur 32 nœuds. Le débit de pointe en multi-nœuds est de 32 Gb/s. **Ces débits élevés permettent l'exploration de nombreuses combinaisons à un niveau de taux d'erreur très faible.** Une partie de ces résultats ont été publiés dans une revue scientifique internationale [1]. **À l'heure actuelle et à notre connaissance, AFF3CT est l'un des simulateurs de codes correcteurs d'erreurs le plus rapide.**

Chapitre 5 - Langage embarqué et dédié à la radio logicielle

Organisation

Ce chapitre présente un nouveau langage embarqué et dédié (*embedded Domain Specific Language*, eDSL) à la radio logicielle (SDR). La première partie décrit les modèles et solutions existants. Elle motive également le besoin d'un nouveau langage dédié à la radio logicielle. Dans une deuxième partie, une description de l'eDSL proposé est donnée et détaillée en deux sous-parties. Dans un premier temps, les composants élémentaires sont présentés, puis, dans un second temps, les composants parallèles sont décrits. La troisième partie se concentre sur l'implémentation des composants présentés précédemment. Entre autres, la technique de duplication des séquences et l'implémentation du pipeline sont discutées. Enfin, la dernière partie présente un cas concret d'utilisation de l'eDSL sur un standard bien répandu dans les communications numériques : la norme DVB-S2. Un émetteur-récepteur entièrement numérique a été conçu en logiciel. La norme DVB-S2 est présentée d'un point de vue applicatif (émetteur et récepteur) et est ensuite évaluée sur une cible CPU spécifique.

Principaux résultats

Les principaux composants de l'eDSL ont été conçus pour répondre aux besoins de la SDR en termes 1) d'expressivité avec des séquences, des tâches et des boucles ; 2) de performance avec la technique de duplication de séquences et la stratégie de pipeline. Nous avons évalué l'eDSL proposé dans un contexte applicatif : l'implémentation logicielle de la norme DVB-S2. **Les résultats démontrent l'efficacité de l'eDSL d'AFF3CT. En effet, la solution proposée répond aux contraintes de temps réel des satellites (30 $\sim$ 50 Mb/s).** Ceci est la conséquence de deux facteurs principaux : 1) les optimisations au niveau des tâches, par exemple un décodeur LDPC rapide a été utilisé ; 2) l'eDSL a un très faible surcoût à l'utilisation. Cela est notamment possible grâce à une implémentation efficace de la technique du pipeline.

Contents

List of Figures

List of Tables

List of Algorithms and Source Codes

List of Acronyms

3G	**Third G**eneration of Mobile Phone Networks
4G	**Fourth G**eneration of Mobile Phone Networks
5G	**Fifth G**eneration of Mobile Phone Networks
A-SCL	**A**daptive **SCL**
A-SSCL	**A**daptive **SSCL**
AFF3CT	**A F**ast **F**orward **E**rror **C**orrection **T**oolbox
AGC	**A**utomatic **G**ain **C**ontrol
AMS	**A**pproximate **M**in-**s**tar
AoS	**A**rray **o**f **S**tructures
API	**A**pplication **P**rogramming **I**nterface
AR	**A**ugmented **R**eality
ASIC	**A**pplication-**S**pecific **I**ntegrated **C**ircuits
AVX	**A**dvanced **V**ector **E**xtensions
AVX-512	**A**dvanced **V**ector **E**xtensions (**512**-bit)
AWGN	**A**dditive **W**hite **G**aussian **N**oise
BCJR	**B**ahl, **C**ocke, **J**elinek & **R**aviv Decoding Algorithm
BB	**B**ase **B**and
BCH	**B**ose, Ray-**C**haudhuri & **H**ocquenghem Channel Codes Family
BE	**B**it **E**rror
BEC	**B**inary **E**rasure Channel
BER	**B**it **E**rror **R**ate
BP	**B**elief **P**ropagation
BP-F	**BP** with **F**looding Scheduling
BP-HL	**BP** with **H**orizontal **L**ayered Scheduling
BP-VL	**BP** with **V**ertical **L**ayered Scheduling
BPSK	**B**inary **P**hase-**s**hift **K**eying
BSC	**B**inary **S**ymmetric **C**hannel
C-RAN	**C**loud **R**adio **A**ccess **N**etwork
CA-SCL	**C**RC-**a**ided **SCL**
CA-SSCL	**C**RC-**a**ided **SSCL**
CCSDS	**C**onsultative **C**ommittee for **S**pace **D**ata **S**ystems
CD	**C**ontinuous **D**elivery
CI	**C**ontinuous **I**ntegration
CPM	**C**ontinuous **P**hase **M**odulation
CPU	**C**entral **P**rocess **U**nit
CRC	**C**yclic **R**edundancy **C**heck
DSL	**D**omain **S**pecific **L**anguage
DSP	**D**igital **S**ignal **P**rocessor
DVB-RCS	**D**igital **V**ideo **B**roadcasting - **R**eturn **C**hannel via **S**atellite
DVB-S2	**D**igital **V**ideo **B**roadcasting for **S**atellite (**Second** Generation)
E-MPA	**E**stimated-**MPA**

ECC	**E**rror **C**orrecting **C**ode
eDSL	**E**mbedded **D**omain **S**pecific **L**anguage
EML-MAP	**E**nhanced **ML-MAP**
FA-SCL	**F**ully **A**daptive **SCL**
FA-SSCL	**F**ully **A**daptive **SSCL**
FE	**F**rame **E**rror
FEC	**F**orward **E**rror **C**orrection
FER	**F**rame **E**rror **R**ate
FPGA	**F**ield **P**rogrammable **G**ate **A**rrays
GA	**G**aussian **A**pproximation
GCC	**G**NU **C** **C**ompiler
GNU	**G**NU's **N**ot **U**NIX
GPP	**G**eneral **P**urpose **P**rocessor
GPU	**G**raphical **P**rocess **U**nit
HIHO	**H**ard **I**nput **H**ard **O**utput
HoF	**H**all **o**f **F**ame
HPC	**H**igh **P**erformance **C**omputing
ICPC	**I**ntel® **C**++ **C**ompiler
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
IoMCT	**I**nternet **o**f **M**ission **C**ritical **T**hings
IoT	**I**nternet **o**f **T**hings
ISA	**I**nstruction **S**et **A**rchitecture
KNC	**Kn**ights **C**orner
KNCI	**Kn**ights **C**orner **I**nstructions
KNL	**Kn**ights **L**anding
LDPC	**L**ow **D**ensity **P**arity **C**heck Channel Codes Family
LLC	**L**ast **L**evel **C**ache
LLR	**L**og **L**ikelihood **R**atio
LTE	**L**ong **T**erm **E**volution
MAP	**M**aximum **a** **P**osteriori
MIPP	**M**y **I**ntrinsics **P**lus **P**lus
ML	**M**aximum **L**ikelihood
ML-MAP	**M**ax-**l**og-**MAP**
MODCOD	**Mod**ulation and **Cod**ing
MPA	**M**essage **P**assing **A**lgorithm
MPI	**M**essage **P**assing **I**nterface
MS	**M**in-**s**um
MSVC	**M**icro**s**oft® **V**isual **C**ompiler
MT19937	**M**ersenne **T**wister **19937**
NMS	**N**ormalized **M**in-**s**um
NOMA	**N**on-**o**rthogonal **M**ultiple **A**ccess
NUMA	**N**on **U**niform **M**emory **A**ccess
OFDM	**O**rthogonal **F**requency-**d**ivision **M**ultiplexing
OMS	**O**ffset **M**in-**s**um
OOK	**O**n-**o**ff **K**eying
OOP	**O**bject-**o**riented **P**rogramming
P-EDGE	**P**olar **E**CC **D**ecoder **G**eneration **E**nvironment
PA-SCL	**P**artially **A**daptive **SCL**
PA-SSCL	**P**artially **A**daptive **SSCL**

PAM	**P**ulse **A**mplitude **M**odulation
PLH	**P**ay **L**oad **H**eaders
PRNG	**P**seudo **R**andom **N**umber **G**enerator
PSK	**P**hase-**s**hift **K**eying
QAM	**Q**uadrature **A**mplitude **M**odulation
QPSK	**Q**uadrature **P**hase-**s**hift **K**eying
R0	**R**ate **0**
R1	**R**ate **1**
RAM	**R**andom **A**ccess **M**emory
REP	**Rep**etition
RF	**R**adio **F**requency
RS	**R**eed & **S**olomon Channel Codes Family
RSC	**R**ecursive **S**ystemactic **C**onvolutionnal
SC	**S**uccessive **C**ancellation
SCL	**S**uccessive **C**ancellation **L**ist
SCMA	**S**parse **C**ode **M**ultiple **A**ccess
SDR	**S**oftware-**d**efined **R**adio
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SISO	**S**oft **I**nput **S**oft **O**utput
SMT	**S**imultaneous **M**ulti-**t**hreading
SNR	**S**ignal-to-**n**oise **R**atio
SoA	**S**tructure **o**f **A**rrays
SoC	**S**ystem **o**n a **C**hip
SPA	**S**um-**p**roduct **A**lgorithm
SPC	**S**ingle **P**arity **C**heck
SPMD	**S**ingle **P**rogram **M**ultiple **D**ata
SSC	**S**implified **SC**
SSCL	**S**implified **SCL**
SSE	**S**treaming **S**IMD **E**xtensions
TDP	**T**hermal **D**esign **P**ower
TNDC	**T**hroughput under **N**ormalized **D**ecoding **C**ost
TPC	**T**urbo **P**roduct **C**odes
TTA	**T**ransport **T**riggered **A**rchitecture
USRP	**U**niversal **S**oftware **R**adio **P**eripheral
VLIW	**V**ery **L**ong **I**nstruction **W**ord
Wi-Fi	**Wi**reless **Fi**delity
WiMAX	**W**orldwide **I**nteroperability for **M**icrowave **Acc**ess
WRAN	**W**ireless **R**egional **A**rea **N**etworks
XOR	E**x**clusive **Or**

Introduction

Digital Communications

Man has sought to communicate from time immemorial. Since then, man has always been seeking for more efficient ways to extend his communication possibilities. Nowadays, with the advent of the Internet, the digital communications represent the last technology advances to communicate world-wide. For instance, digital communications enable both live video transmission and the use of a messaging system. With the growing number of users and needs, the digital communication systems are the subject of an important area of research. New digital communication systems have to be able to match high throughput and low latency constraints as well as acceptable energy consumption levels.

Traditionally, digital communication transmitters and receivers are implemented in hardware, on dedicated chips. The required signal processing algorithms are often very specific and repetitive. Thus, they are good candidates for specialized architectures. However, with the growing number of use cases and telecommunication standards, these algorithms are evolving and are becoming more and more heterogeneous. In this context, it becomes interesting to consider software implementations on generic architectures. This type of programmable architectures is available in computers and is commonly referred as the Central Process Unit (CPU). The CPUs are General Purpose Processors (GPP) that can adapt to various types of algorithms.

Computer Architecture

Improving the computational and the energy efficiency of these processors is one of the main concern in computer science. As they are largely adopted for many use cases, the CPUs take advantage of the best manufacturing processes. Thanks to their pipelined architecture they are able to reach very high processing frequencies ranging from 1 to 4 GHz. They also come with dedicated memory caches that enable efficient spatial and temporal reuse of data. Nowadays, the computational efficiency of the CPUs relies on two main parallel techniques. The first one is the multi-core architecture: it consists in duplicating the hardware of the "CPU" in multiple instances called cores. These cores are mostly independent from each other. They are packaged together in the same chip (called the CPU) and generally they share a fast memory: the last level cache. The second parallel technique is the vectorized instructions. These types of instructions are available in each core and are able to perform the same operation on a chunk of data. This is also known as the Single Instruction Multiple Data (SIMD) architectural model.

From an energy point of view, it is clear that CPUs are not directly competitive compared to dedicated architectures. Their large number of instructions enables efficient implementations of many algorithms but this is also a limitation when targeting specific applications. Many transistors are unused and consume a non-negligible amount of energy. On the other hand, the main strength of the GPP architectures comes from their abilities to be used programmatically

with high level languages. Consequently, the time required to implement new algorithms is much shorter on GPPs than on dedicated hardwares. However, even with reduced implementation time, it is still challenging to design algorithms that take effectively advantage of the CPUs parallelism levels.

Hardware Abstraction and Software

The ever growing complexity of processors motivates new hardware description level abstractions. Even if it is still possible to write assembly codes, one should agree that this is not adapted to real-size applications. Moreover, in general the designers of an application are not familiar with the specificities of the CPU architectures. Thus, it becomes important to propose new models (or abstractions) on top of the hardware. It enables the efficient use of processors to the largest number of people. To this purpose, dedicated compilers, languages and libraries are an important research area in computer science.

In the design of digital communication systems, it is now common to rely on software implementations for the evaluation and the validation of signal processing algorithms. These evaluation and validation steps consist in the simulation of the whole communication system. Generally these type of simulations are implemented by signal processing experts with high level programing languages like MATLAB® or Python. However, with the growing complexity of the digital communication systems, these simulations are becoming more and more compute intensive. Using high level programming languages is a limiting factor because it can lead to large restitution times (from days to weeks). Thus, high performance implementations based on lower level programming languages are considered.

Moreover, software implementations are also considered for real-time uses. Their flexibility and reduced time to market are becoming more and more attractive. Indeed, dedicated hardware solutions require specific skills and are achieved by electronics specialists. In general, signal processing experts do not focus on implementing efficient software or hardware solutions. The dialog between the two communities is not always simple as they have different concerns.

The purpose of this thesis is to ease the overall design of digital communication systems, from the conception to the implementation. Dedicated tools and interfaces are proposed to help the signal processing experts to design fast software implementations. In general, this type of software implementations are a good start to better understand the algorithms hotspots. From this point, electronics specialists can improve proposed software solutions and, if necessary, design adapted hardware implementations.

Contributions

In this thesis we propose to study the most time consuming algorithms of digital communication systems, to adapt and optimize them on General Purpose Processors (GPPs) like the CPUs. Most of the current digital communication standards require the implementation of such algorithms. The long simulation times and the real-world application requirements make it desirable to have portable, flexible, high throughput and low latency implementations. The proposed high performance implementations are shown to be competitive with the state-of-the-art ones. Contrary to the previous works, this thesis strives to extract generic methodologies and strategies common to the majority of the signal processing algorithms. The proposed implementations try to be as flexible as possible without sacrificing too much the performance.

The signal processing algorithms come with various characteristics. Thus, it is of interest to be able to manage this algorithmic heterogeneity. In this work, to enable code reuse, similarities are identified into this zoo of algorithms. The various implementations have been packaged, categorized and organized in one single software library, namely AFF3CT. These implementations cohabit together thanks to well-defined interfaces and an adapted software architecture based on the Object-Oriented Programming (OOP) paradigm.

Another important concern of this work is the ability to reproduce the scientific results. Indeed, all the proposed implementations are regrouped in AFF3CT which is an open-source software. Specific strategies have been operated to minimize the possible regressions based on the digital communication systems characteristics. These non-regression strategies are automated. They ensure that the source code remains stable even if many contributors are working together.

These contributions have been the topic of several scientific publications in both the computer science and the signal processing communities. They are listed at the end of the manuscript in the "Personal Publications" section. As a convention in the document, the numeric citations are contributions of this thesis while the alphabetic citations refer to other works in the literature.

Dissertation Organization

This dissertation is organized in five chapters. The first chapter describes the context and details the objectives. The next chapters present our contributions.

In Chapter 1, the digital communication systems are detailed. Then, the most time consuming part of these systems is presented, namely the channel decoders. After that, the applicative contexts of this thesis are defined. The two main ones are the functional simulation and the Software-Defined Radio (SDR). The functional simulation enables the evaluation and the validation of different digital communication systems while the SDR corresponds to the real-time execution of these systems in software. Finally, the main problematics are exposed.

In Chapter 2, new efficient implementations of the decoders are proposed. First, an overall portable methodology is detailed to meet the high throughput constraint required by both the simulations and the real-time systems. This methodology is based on the Single Instruction Multiple Data (SIMD) model implemented in most of the current CPUs. Depending on how the CPU SIMD instructions are used, it is possible to maximize the throughput or the latency of the implemented decoding algorithms. Then, specific optimized implementations are detailed for each decoding algorithm. These implementations focus on maximizing the flexibility, high throughput and low latency. Depending on the implementations, some compromises have to be made and some of these characteristics can be maximized unbeknownst to others.

In Chapter 3, AFF3CT, our toolbox dedicated to the forward error correction (FEC) algorithms is presented. AFF3CT is unique in the domain and it is composed by many algorithm implementations (including those presented in Chapter 2). AFF3CT is the software that enables the signal processing algorithms heterogeneity thanks to a robust software architecture based on well-defined and coherent interfaces. It enables reproducibility of the results as it is open-source and extensively tested. AFF3CT also contains a parallel functional simulator and enables extensive exploration/validation of existing or new algorithms on a large combination of parameters.

In Chapter 4, the efficient algorithm implementations proposed in Chapter 2 are evaluated and compared with the state-of-the-art. The FEC software decoders hall of fame is introduced to summarize and to compare the proposed contributions with previous works in the literature. Some metrics are defined for ease of comparison. These metrics focus on normalized throughput, proper use of hardware and energy efficiency. Finally, the AFF3CT simulator efficiency is demonstrated on various multi-core CPUs and on a multi-node cluster.

In Chapter 5, a new embedded Domain Specific Language (eDSL) for the SDR is presented. The AFF3CT software suite is enriched with new blocks dedicated to the efficient implementation of real-time digital communication systems on multi-core CPUs. These blocks enable automatic parallelism. As an example of use, a full physical layer of the DVB-S2 standard has been implemented. All the digital processing are performed with AFF3CT while the radio frequency communications is achieved with Universal Software Radio Peripherals (USRPs). The results match the satellite real-time constraints.

1 Context and Objectives

This chapter introduces the context of digital communication systems. Its purpose is to define notions that will be reused in the manuscript and to give a general view and defines the main objectives of this thesis. The first section presents the principle of the digital communication systems with its different parts: transmitter, channel and receiver. The most common metrics used in digital communications are also introduced. The second section details the channel model and the digital modulation used all along the manuscript. A characterization of the signal-to-noise ratio is given as well as the notion of probability at the output of the channel and the demodulator. The third section introduces the channel codes considered in this manuscript: namely the LDPC, the polar and the turbo codes. The corresponding encoding and decoding processing are detailed for each channel code family. In the fourth section, three applicative contexts are detailed for the considered channel codes. The functional simulation enables the design and the validation of a coding scheme. The software-defined radio is a radio communication system where components are implemented by means of software. Then, the sparse code multiple access mechanism is presented as it is a promising solution for massive connectivity in future mobile networks. The fifth section gives the main problematics of the thesis.

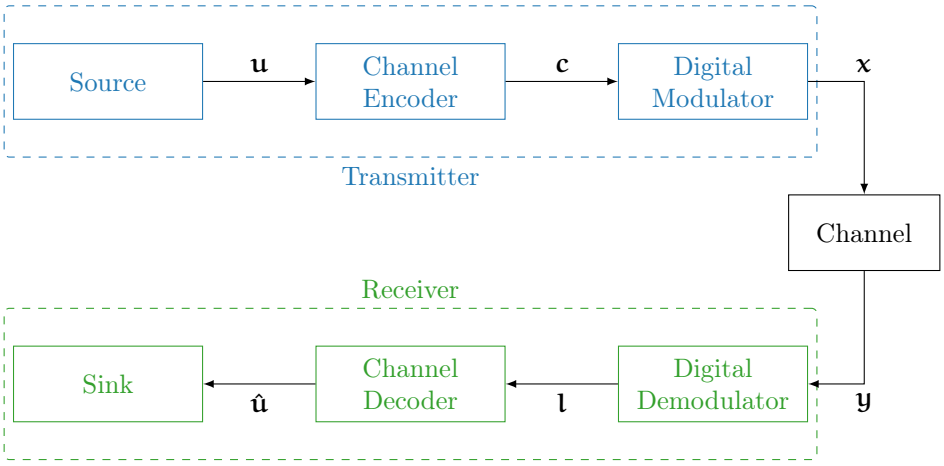


Figure 1.1 – Digital communication chain.

1.1 Digital Communication Systems

It is now commonplace to state that Humanity has entered the era of communication. By 2025, there should be more than 5 billion smart-phones in circulation worldwide. Moreover, all kinds of objects will increasingly use communication technology, to exchange information in the *Internet of Things* (IoT), for instance. Despite their heterogeneity, all communication systems are based on a common abstract model proposed by Claude Shannon. In his seminal paper [Sha48], he proposed to model a communication system with five major components: an information source, a transmitter, a channel, a receiver and a destination. This model was later refined as shown in Figure 1.1. The source produces a digital message $\mathbf{u}$ to be transmitted (sequence of bits). The channel encoder transforms it in a codeword $\mathbf{c}$ to make it more robust to errors. In order to make possible the information transmission through the channel, it is necessary to shape the data stream. For instance, in the case of wireless communication, this stream must be represented by a high-frequency signal in order to be transmitted by a reasonably sized antenna. This is the role of the digital modulator which produces a vector of symbols $\mathbf{x}$. The channel alters the signal with some noise and distortions ($\mathbf{y}$). On the receiver side, the components perform the inverse operations to retrieve the decoded message $\hat{\mathbf{u}}$. If no errors occurred during the transmission or if there is errors but they have been corrected, $\hat{\mathbf{u}} = \mathbf{u}$.

In the next sections and chapters, we will focus on channel coding because it comes with algorithms that have the highest computational complexity in the digital communication systems. In channel coding, also known as *forward error correction* (FEC), K information bits are encoded in the transmitter. It results in a codeword $\mathbf{c}$ of N bits. $\mathsf{P} = \mathsf{N} - \mathsf{K}$ is the number of redundancy bits added as additional information and $\mathsf{R} = \mathsf{K}/\mathsf{N}$ is the code rate. The higher the code rate R is, the lower the number of bits P is. The performance of this scheme is measured by estimating the residual error rate at the sink. It is possible to observe two different rates: 1) the Bit Error Rate (BER); 2) the Frame Error Rate (FER). The BER is calculated considering the K information bits independently, for instance a 10^{-3} BER means that there is an average of one binary error per thousand information bits transmitted. The FER is computed considering the entire frame, if there is at least one wrong bit in the current frame, it will be counted as one erroneous frame. A 10^{-2} FER means that there is an average of one frame error per hundred frame transmitted. These rates depend on many factors: the noise of the channel, the modulation type, the code type, the code rate R, etc. The lower the bit and frame error rates are, the higher the correction power of the system is.

1.2 Channel Model

In this thesis, only causal transmission channels without memory effect and stationary are considered. In other words, the output of the channel at time t only depends on its input at time t. In order to describe the disturbance applied to the message $\mathbf{x}$ passing through the transmission channel, different models can be used. However, in the literature, the selected model is often the Additive White Gaussian Noise (AWGN) channel. In particular, this channel well models the thermal noise which is one of the sources of noise that is always present on the receiver side. This section presents the AWGN channel concepts and introduces the Binary Phase-Shift Keying modulation/demodulation that is employed all along the manuscript.

In the AWGN channel, the law binding the y_i output to its x_i input is of the form $y_i = x_i + n_i$ with N_{chn} an independent and identically distributed variable according to a normal (or Gaussian) law centered in zero and of variance $\sigma^2 = N_0/2$. So, we have $N_{chn} \simeq \mathcal{N}(0, \sigma^2)$ and:

$$P(y_i|x_i) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y_i - x_i)^2}{2\sigma^2}\right). \tag{1.1}$$

To estimate the correction power of a channel code it is very common to vary the Signal-to-Noise Ratio (SNR). On the AWGN channel the SNR is generally given by E_b/N_0 (in dB). E_b corresponds to the average energy per information bit. It can also be given by E_s/N_0 (in dB) where E_s corresponds to the average energy per transmitted symbol. A symbol is a binary or a non-binary quantity, so it can be represented by one or more bits. E_s/N_0 can be deduced from E_b/N_0 as follows:

$$\frac{E_s}{N_0} = \frac{E_b}{N_0} + 10.\log(R.b_S), \tag{1.2}$$

where R is the code rate and b_S is the number of bits per transmitted symbol x_n. b_S depends on the modulation order. If a binary modulation is used, then $b_S = 1$. The channel variance is:

$$\sigma = \sqrt{\frac{1}{2 \times 10^{\frac{E_s}{N_0}/10}}}. \tag{1.3}$$

An important characteristic of a channel is its capacity [Sha48]. The capacity represents the maximal quantity of information that the canal can transport. In other words, it is impossible to find a coding scheme that transports more information than the channel capacity. From this capacity it is possible to deduce Shannon's limit. This limit is the asymptotic SNR in E_b/N_0 (dB) which cannot be improved with any channel code. When R tends towards zero it can be shown that Shannon's limit is -1.59 dB. This means that, for an AWGN channel, no system can reliably transmit information at an SNR of less than -1.59 dB.

In the next chapters of the manuscript, the AWGN channel will mainly be associated with a Binary Phase-Shift Keying (BPSK) modulation ($b_S = 1$). With this modulation, each binary value $c_i \in \{0, 1\}$ is associated to a real value $x_i \in \{1, -1\}$. The l outputs estimated by the digital demodulator can be given in the form of a Log Likelihood Ratio (LLR). Their sign determines for each channel output data $y_i \in \mathbf{y}$ the most likely binary input $c_i \in \mathbf{c}$. The absolute value

corresponds to the degree of reliability of the information. The mathematical expression of l_i is:

$$l_i = \log\left(\frac{P(y_i|c_i = 0)}{P(y_i|c_i = 1)}\right).$$

1.3 Channel Codes

After Shannon's work, researchers have designed new coding/decoding schemes to approach Shannon's theoretical limit increasingly closer. Indeed, recent progresses managed to design practical codes performing very close to that limit. These codes are already integrated in current communication systems. They are usually classified in two main families: block codes and convolutional codes. The block codes generate the redundancy by packets of data while the convolutional codes compute the redundancy bit by bit on the data stream. The purpose of this section is to introduce the most used channel code families.

The convolutional codes have been introduced by Peter Elias in 1955 [Eli55]. The objective was to propose an alternative to the block codes in term of codeword length flexibility: theoretically the length of a convolutional code is infinite. The coding scheme is made in a way that the output depends on the current input and on the inputs before. This type of code has been used by the NASA for satellite communications for instance.

The Raj **B**ose, D. K. Ray-**C**haudhuri and Alexis **H**ocquenghem (BCH) codes are block codes discovered in the late 1950s [Hoc59, BR60]. They are algebraic codes built from a polynomial. This results in low complexity decoding algorithms. These codes are used in the CDs, DVDs and SSDs. In modern coding schemes, they are often concatenated to other codes in order to improve their correction powers especially when there are erroneous frames with a small number of erroneous bits. This is the case in the DVB-S2 standard for instance.

The Irving S. **R**eed and Gustave **S**olomon (RS) codes have been proposed in 1960 [RS60]. Like the BCH codes they are algebraic codes. But unlike the BCH codes, the RS codes are based on symbols instead of bits (non-binary codes). The RS codes are used in many standards (CD, DVD, Blu-ray, ADSL, DVB-T, etc.).

The Low-Density Parity-Check (LDPC) codes are linear block codes. They have been discovered by Robert G. Gallager in 1962 [Gal62]. Unfortunately, at the time of their discovery, the computational power available in the transceivers was not sufficient to decode them. Later, in 1995, the LDPC codes have been re-discovered by David MacKay [MN95]. Nowadays, they are used in many digital communication standards such as Wi-Fi, WiMAX, WRAN, 10Gbps Ethernet, DVB-S2, CCSDS, 5G data transport, etc.

The turbo codes have been discovered by Claude Berrou in 1993 [BGT93] and have been used in many digital wireless communication standards since (3G, 4G, DVB-RCS2, CCSDS, etc.). The particularity of the turbo codes is to be composed by two convolutional sub-codes. In other terms, the turbo codes are a parallel concatenation of two convolutional codes.

The Turbo Product Codes (TPC) are block codes, they have been invented by Peter Elias in 1954 [Eli54] while the first efficient decoding algorithm has been discovered later in 1994 by Ramesh Pyndiah [Pyn+94]. A TPC is a form of serial concatenation of two block codes, it results in a matrix where one code can be read from the columns and the other one from the rows. The TPC are notably used in the WiMAX standard and also in some optical communication systems.

The polar codes are linear block codes like the LDPC. They have been invented by Arıkan in 2009 [Arı09]. For the first time, they are present in the 5G standard (control channels). The particularity of these codes is that they are the only ones for which it has been mathematically demonstrated that they reach Shannon's limit (considering an infinite codeword length).

Many other codes exist like Hamming codes, Plotkin codes, Gilbert codes, Golay codes, Reed-Muller codes [Mul54, Ree54], the raptor codes [Sho04], etc. The purpose of this section is not to be exhaustive but to give a representative overview of the FEC domain. This thesis will focus on a subset of these codes, namely the LDPC codes, the polar codes and the turbo codes. These codes have been selected because they are known to be channel capacity-approaching and well-spread in the current digital communication standards. Moreover, these code families lead to high computational complexity decoders that are challenging to implement.

1.3.1 Prerequisites

Table 1.1 – Elementary operations in GF_2 (logical *exclusive or* and logical *and*).

a	b	$a \oplus b$	ab
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

In all the presented coding schemes, only binary codes are considered. In this case, a bit can be represented in a Galois field of two elements $\{0, 1\}$ denoted as GF_2. A block code is an application g of GF_2^K in GF_2^N with $K < N$. There are 2^K codewords $\mathbf{c}$. The two operations used to generate a codeword are the addition and the multiplication. In GF_2, the addition is equivalent to a logical *exclusive or* ($\oplus$) and the multiplication is equivalent to a logical *and* (see Table 1.1).

In this chapter, all the decoding algorithms are working on *soft* information. This means that the decoding input is a vector of N likelihoods in the form of LLRs. Each LLR is a real value. Depending on the implementation it can be a floating-point or a fixed-point number. It results in more complex operations than for the encoding process. On the decoding side and in the logarithmic domain, the $\oplus$ operator can become the $\boxplus$ operator, it is defined as follow:

$$l_a \boxplus l_b = 2\tanh^{-1}\left(\tanh\left(\frac{l_a}{2}\right).\tanh\left(\frac{l_b}{2}\right)\right). \tag{1.4}$$

This is the main reason why, in channel coding, the decoders are systematically more compute intensive than the encoders. In the logarithmic domain, the multiplication becomes a simple addition.

1.3.2 Low-density Parity-check Codes

1.3.2.1 Coding Scheme

A parity-check constraint is an equation that links a set of bits: when all the bits of a parity-check constraint are added together the result has to be zero. For instance, if we consider a message $\mathbf{u} = [u_0, u_1, u_2, u_3]$ ($K = 4$), then it is possible to encode the information message $\mathbf{u}$ in a codeword $\mathbf{c}$ of size $N = K + 1 = 5$: $\mathbf{c} = [u_0, u_1, u_2, u_3, p_0]$. The parity-check constraint $\mathcal{C}_0$ is

then: $u_0 \oplus u_1 \oplus u_2 \oplus u_3 \oplus p_0 = 0$ ($\mathcal{C}_0$) with p_0 the parity bit ($P = N - K = 1$). To encode the message $\mathbf{u}$ and produce the codeword $\mathbf{c}$, a generator matrix $\mathcal{G}$ (or a linear application) has to be defined like this: $\mathbf{c} = \mathbf{u} \times \mathcal{G}$ with

$$\mathcal{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}.$$

$\mathbf{u} \times \mathcal{G} = [u_0, u_1, u_2, u_3, u_0 \oplus u_1 \oplus u_2 \oplus u_3] = \mathbf{c}$, so $p_0 = u_0 \oplus u_1 \oplus u_2 \oplus u_3$ as defined by the parity-check constraint $\mathcal{C}_0$. The proposed generator matrix $\mathcal{G}$ is composed by the identity matrix on the four first columns and by the parity-check constraint in the last column. The consequence of the presence of the identity matrix is that the generated codeword contains the initial information bits u_0, u_1, u_2, and u_3. In this case, the encoding process is *systematic*.

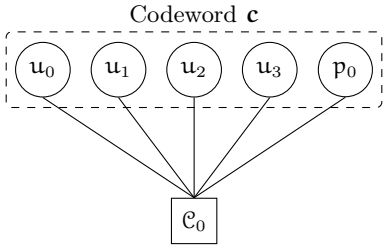


Figure 1.2 – Representation of the $\mathcal{C}_0$ parity-check constraint on a Tanner graph.

One can note that a parity-check constraint can also be represented with a Tanner graph as shown in Figure 1.2. It is also possible to define a matrix of parity-check constraints namely $\mathcal{H}$. In this case, there is only one constraint ($\mathcal{C}_0$), so $\mathcal{H}$ is a one-dimension matrix (or a vector) of size N: $\mathcal{H} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \end{bmatrix}$. An important property of the $\mathcal{H}$ matrix is that it must satisfy: $\mathcal{G} \times \mathcal{H}^\mathsf{T} = \mathbf{0}$.

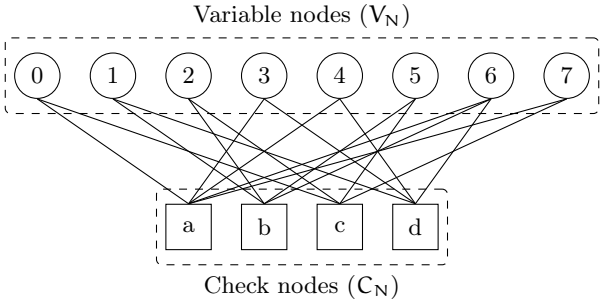


Figure 1.3 – Parity-check constraints of an LDPC code on a Tanner graph.

The construction of a Low-Density Parity-Check (LDPC) code is based on the combination of several parity-check nodes. Figure 1.3 is an example of a LDPC code with four parity-check constraints denoted as $\mathtt{a}$, $\mathtt{b}$, $\mathtt{c}$ and $\mathtt{d}$. The parity-check constraints are also known as the *check nodes* (C_N). The *variable nodes* (V_N) are the bits of the LDPC codeword. The parity-check matrix corresponding to the Figure 1.3 Tanner graph is:

$$\mathcal{H} = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

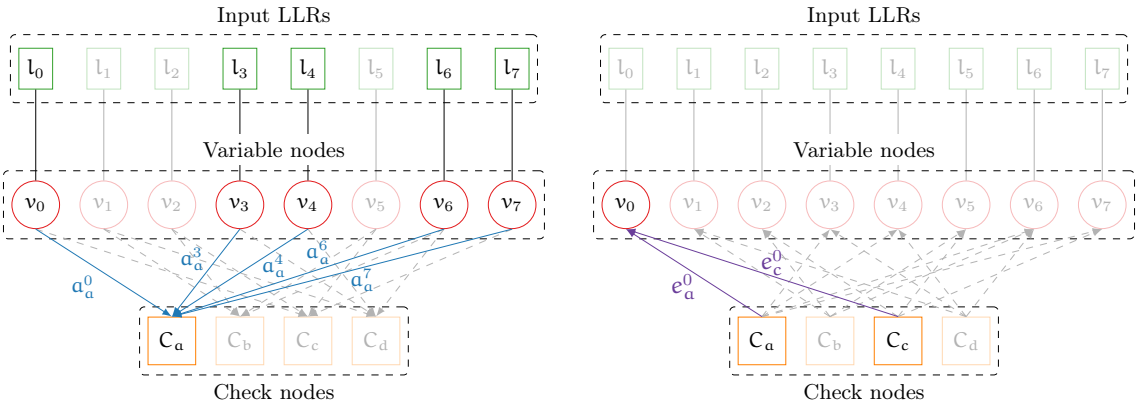
The $\mathcal{H}$ parity matrix of an LDPC code has to be a low-density matrix. The example shown in Figure 1.3 is here to help the comprehension and is not a real LDPC code: indeed, the corresponding $\mathcal{H}$ matrix is not sparse.

1.3.2.2 Belief Propagation Decoding Algorithm

The bit $\hat{u}_n$ corresponding to the input LLR l_n of a parity-check code can be decoded as follow: $\hat{u}_n = h_d\left(l_n + \sum_{j \neq n} l_j\right)$, with h_d the hard decision function that returns 0 if the LLR is positive and 1 otherwise. For instance, considering the parity-check code in Figure 1.2, $\hat{u}_0 = h_d\left(l_0 + (l_1 \boxplus l_2 \boxplus l_3 \boxplus l_4)\right)$, $\hat{u}_1 = h_d\left(l_1 + (l_0 \boxplus l_2 \boxplus l_3 \boxplus l_4)\right)$, etc.

In LDPC codes, there is more than one parity-check node. It is then possible to compute all the check nodes connected to a variable node and to store the result in a vector $\boldsymbol{\nu}$. Each LLR $\nu_n \in \boldsymbol{\nu}$ corresponds to one variable node. For instance, considering Figure 1.3, V_0 is connected to C_a and C_c. So its LLR value can be computed as follow: $\nu_0 = e_0 + e_1 = (l_3 \boxplus l_4 \boxplus l_6 \boxplus l_7) + (l_2 \boxplus l_5 \boxplus l_7)$, where e_0 and e_1 are the extrinsic informations computed from C_a and C_c, respectively. The decoded bits can be decided from the channel and the variable node LLR values: $\hat{u}_n = h_d\left(l_n + \nu_n\right)$.

In the Belief Propagation (BP) decoding algorithm, there are many iterations (5 to 100) between the variable nodes and the check nodes, before to decide the decoded bits $\hat{\mathbf{u}}$. In the first iteration, the a priori information $\mathbf{a}$ sent to the check nodes is directly the channel values $\mathbf{l}$. But, in the next iterations, the a priori information $\mathbf{a}$ is updated with the variable nodes values $\boldsymbol{\nu}$. To avoid direct auto-confirmation issues, the up-coming extrinsic LLR is systematically subtracted from the propagated message.



(a) Check nodes update.

(b) Variable nodes update.

Figure 1.4 – Illustration of the belief propagation algorithm on a Tanner graph.

Figure 1.4 illustrates a single BP iteration. First, the check nodes are computed from the messages m_j^i where i is the index of the variable nodes and j is the index of the check nodes. In the example, the C_a check node computes $a_a^0 \boxplus a_a^3 \boxplus a_a^4 \boxplus a_a^6 \boxplus a_a^7$, where $a_a^0 = l_0 + \nu_0 - e_a^0$, $a_a^3 = l_3 + \nu_3 - e_a^3$, etc. During the first iteration $\boldsymbol{\nu}$ and $\boldsymbol{e}$ are initialized to 0. Then, when all the check nodes have been computed, it is possible to estimate the new values of the ν_n variable nodes from the sum of the incoming extrinsic messages e_j^n. In the example, $\nu_0 = e_a^0 + e_c^0$, where $e_a^0 = a_a^3 \boxplus a_a^4 \boxplus a_a^6 \boxplus a_a^7$ and $e_c^0 = a_c^2 \boxplus a_c^5 \boxplus m_c^7$. When all the variable nodes have been updated, it is then possible to update the check nodes and so on.

There are many variants of the BP algorithm. In the previous explanation, during an iteration, all the check nodes are computed first. Then, all the variable nodes are updated. This is called *flooding* (BP-F) scheduling of the computations [MN95]. However it is possible to schedule the computations differently. In the *horizontal layered* (BP-HL) scheduling [Yeo+01], when a check node is evaluated, all the connected variable nodes are updated without waiting the computation of all the check nodes. In the *vertical layered* (BP-VL) scheduling [ZF02], the check nodes corresponding to a variable node are evaluated and the current variable node is updated. The vertical layered scheduling traverses the variable nodes first while the horizontal layered scheduling processes the check nodes first. In general, the layered scheduling (vertical and horizontal) enables to converge faster (in less iterations than the flooding) to a valid codeword.

In the previous example, the rules to update the variable nodes are based on the $\boxplus$ operation. In the literature, this type of update rules is called the *Sum-Product Algorithm* (SPA) and was first introduced by Gallager in 1962 [Gal62]. The SPA results in very good BER/FER decoding performance. However, this comes at the cost of a high computational complexity. To reduce the computational complexity on the $\boxplus$ operator it is possible to approximate it as follow:

$$l_a \boxplus l_b = 2\tanh^{-1}\left(\tanh\left(\frac{l_a}{2}\right).\tanh\left(\frac{l_b}{2}\right)\right) \approx \text{sign}(l_a.l_b).\min(|l_a|,|l_b|). \tag{1.5}$$

This variant is called the *Min-Sum* (MS) [FMI99]. The costly tanh functions are replaced by efficient sign and min operations. However, MS computations negatively affect the correction performance. To compensate the performance loss, the optimized *Offset Min-Sum* (OMS) and *Normalized Min-Sum* (NMS) approximations have been proposed in [CF02]:

$$l_a \boxplus l_b = 2\tanh^{-1}\left(\tanh\left(\frac{l_a}{2}\right).\tanh\left(\frac{l_b}{2}\right)\right) \approx \alpha \times \Big(\text{sign}(l_a.l_b).\min(|l_a|,|l_b|) + \lambda\Big), \tag{1.6}$$

where α is a normalization factor of the NMS update rules and λ is an offset of the OMS update rules.

1.3.3 Polar Codes

1.3.3.1 Coding Scheme

A polar code (N, K) is a linear block code of size $N = 2^m$, with N the first natural number higher than K. The $\mathcal{G}$ generator matrix of a polar code can recursively be defined by the m^{th} Kronecker power of $\mathcal{K} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$, denoted as $\mathcal{G} = \mathcal{K}^{\otimes m} = \begin{bmatrix} \mathcal{K}^{\otimes m-1} & 0_{m-1} \\ \mathcal{K}^{\otimes m-1} & \mathcal{K}^{\otimes m-1} \end{bmatrix}$, composed by N rows and N columns. Unlike for the LDPC codes, the $\mathbf{u}$ input message cannot be directly multiplied by $\mathcal{G}$. Indeed, $\mathcal{G}$ is a square matrix of dimension N. So, the polar coding scheme defines an $\mathcal{F}$ function that adds zeros in $\mathbf{u}$ until its size reaches N bits ($\boldsymbol{\nu} = \mathcal{F}(\mathbf{u})$). If we suppose a $(8, 4)$ polar code, $\mathbf{u} = [u_0, u_1, u_2, u_3]$ is composed of 4 information bits. Lets apply the $\mathcal{F}$ function on $\mathbf{u}$: $\mathcal{F}(\mathbf{u}) = [0, 0, 0, u_0, 0, u_1, u_2, u_3] = \boldsymbol{\nu}$. There is N output bits in $\boldsymbol{\nu}$. The extra zeros are called the *frozen bits*. Their positions in $\boldsymbol{\nu}$ are selected to be on the less reliable indexes. In other terms, the information bits occupy the most reliable positions in $\boldsymbol{\nu}$. The frozen bits represent the P parity bits. In this thesis, the Gaussian Approximation (GA) method is used to determine the position of the frozen bits [Tri12]. To summarize, the polar encoding process can be defined as follow: $\mathbf{c} = \mathcal{F}(\mathbf{u}) \times \mathcal{G} = \boldsymbol{\nu} \times \mathcal{G}$.

$$\mathcal{G}_{N=2} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

$$\mathcal{G}_{N=4} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$\mathcal{G}_{N=8} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

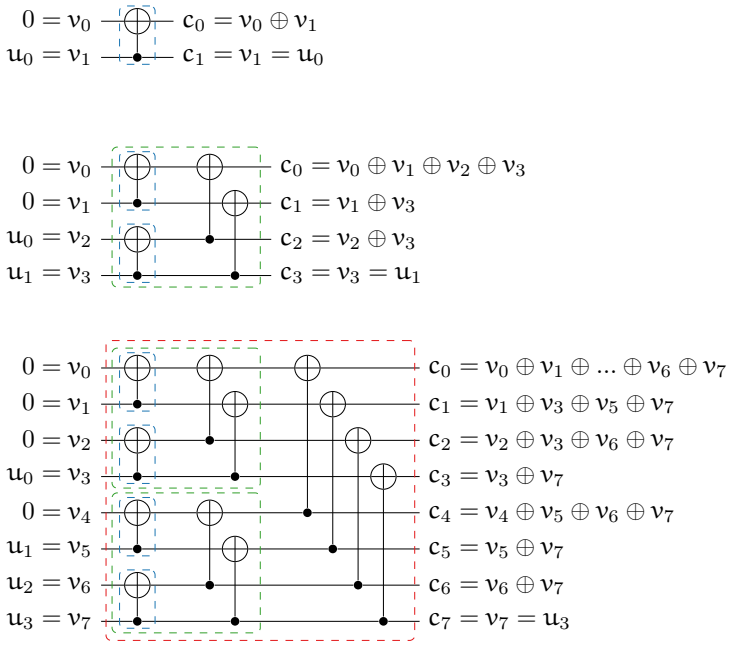


Figure 1.5 – Polar encoding process for $N \in \{2, 4, 8\}$ and $R = 1/2$.

Figure 1.5 presents $\mathcal{G}$ generator matrices depending on N and their associate encoding schemes described with factor graphs. The recursive structure of the polar codes is represented by the dashed rectangles in the factor graphs. For instance, when $N = 8$, the encoder is composed of two $N = 4$ sub-encoders. Each $N = 4$ sub-encoder is itself composed of two $N = 2$ sub-encoders. The polar code are not necessarily systematic.

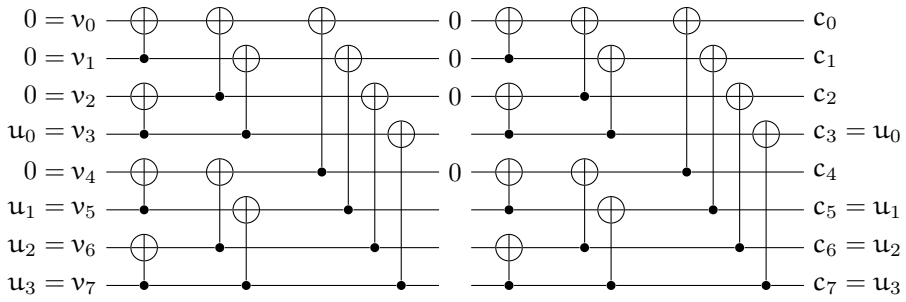


Figure 1.6 – Systematic polar encoder for $N = 8$ and $R = 1/2$.

In 2011, Arıkan proposed a systematic coding scheme for the polar codes [Arı11]. The idea is to apply a pre-encoding step before the Kronecker transformations. Figure 1.6 shows the systematic polar encoder for $N = 8$. The systematic encoding scheme can be expressed as: $\mathbf{c} = \mathcal{F}'\Big(\mathcal{F}(\mathbf{u}) \times \mathcal{G}\Big) \times \mathcal{G}$, with $\mathcal{F}'$ the function that reinitializes the frozen bits to zero after the first encoding. The systematic encoding is possible because of the characteristics of the $\mathcal{G}$ generator polar matrices: $\mathcal{G} \times \mathcal{G} = \mathbf{I}$. In other terms, $\mathcal{G}$ is invertible and its inverse is itself. A direct consequence of this property is that one can encode from left to right or from right to left: the generated codeword $\mathbf{c}$ will be the same. This is why the factor graphs proposed in Figure 1.5 and Figure 1.6 are not directed.

It is also possible to represent the polar encoding process with a binary tree structure. Figure 1.7 shows the binary tree representation of an $(8, 4)$ polar encoder. The leaf nodes represent the initial bits from the $\boldsymbol{\nu}$ vector. The bits in black are the information bits $\mathbf{u}$ and the

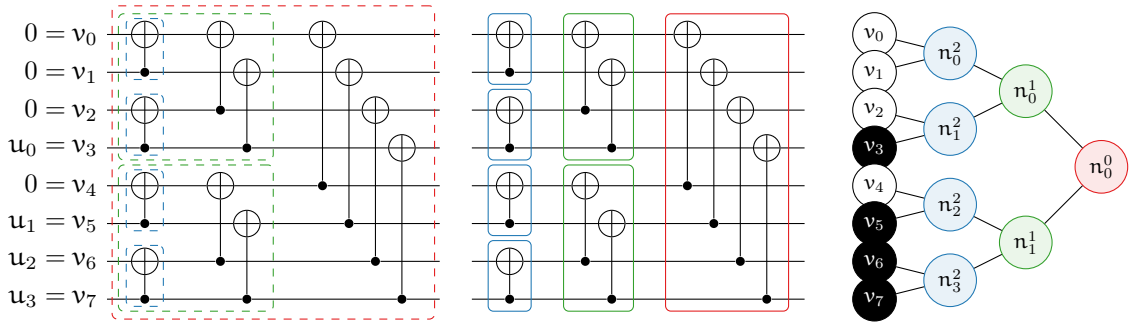


Figure 1.7 – Tree representation of a polar encoder for $N = 8$ and $R = 1/2$.

white bits are the frozen bits. Two by two the initial bits are bound to a father node n_x^2 where x is the index of the node in the layer 2. In general, a node is denoted by n_x^d where d is the depth (or layer) in the binary tree. The blue nodes compute the sub-graphs delimited by the solid blue rectangles (one XOR per node). The green nodes compute the sub-graphs delimited by the solid green rectangles (two XORs per node). The red node computes the sub-graph delimited by the solid red rectangle (four XORs per node).

1.3.3.2 Successive Cancellation Decoding Algorithm

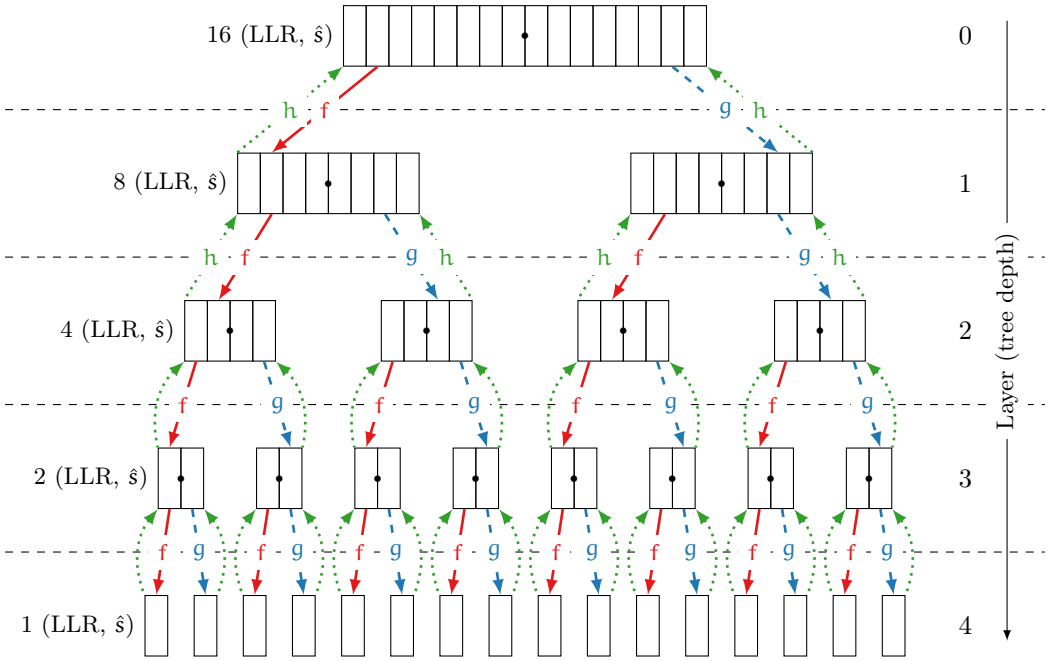


Figure 1.8 – Full SC decoding tree ($N = 16$).

A first decoding algorithm, called the Successive Cancellation (SC) decoding algorithm, has been introduced by Arıkan [Arı09]. It can be seen as the traversal of a binary tree starting from the root node. For a code length $N = 2^m$, the corresponding tree thus includes $m + 1$ node layers, indexed from $d = 0$ (root node layer) down to $d = m$ (leaf nodes layers). As the tree is initially full, each layer d contains 2^d nodes, each node of that layer d containing 2^{m-d} LLRs (λ) and 2^{m-d} binary values denoted as *partial sums* ($\hat{s}$). At the decoder initialization, LLR values received from the channel (l) are stored in the root node. Then, the decoding process performs a pre-order traversal of the tree. When a node is visited in the downward direction, LLRs of the node are updated. In the upward direction, partial sums are updated. Figure 1.8 summarizes the

computations performed in both directions. The update functions are:

$$\begin{cases} \lambda_c & = & f(\lambda_a, \lambda_b) & = & \lambda_a \boxplus \lambda_b \approx \text{sign}(\lambda_a.\lambda_b).\min(|\lambda_a|, |\lambda_b|) \\ \lambda_c & = & g(\lambda_a, \lambda_b, \hat{s}) & = & (1-2\hat{s})\lambda_a + \lambda_b \\ (\hat{s}_c, \hat{s}_d) & = & h(\hat{s}_a, \hat{s}_b) & = & (\hat{s}_a \oplus \hat{s}_b, \hat{s}_b). \end{cases} \tag{1.7}$$

The f and g functions both generate a single LLR. The h function provides a couple of partial sums. The f function is the Min-Sum approximation of the $\boxplus$ operation described in Equation 1.5. In Polar decoding using an MS approximation does not significantly impact the decoding performance. Thus, the MS approximation is widely applied.

Before recursively calling itself on the left node, the algorithm applies the f function, respectively. Before calling itself on the right node the g function is applied. At the end (after the recursive call on the right node) the h function is applied. The f and g functions operate on the LLRs (read only mode) from the current node n_i in order to produce the new LLR values into left and right n_{i+1} nodes, respectively. The h function reads the bits from the left and right n_{i+1} nodes in order to update the bit values of the n_i node. The $\boldsymbol{\lambda}$ LLRs in the leafs are converted in the $\hat{\mathbf{s}}$ bits with the hard decision function $\hat{s}_n = h_d(\lambda_n)$.

Leaf nodes are of two kinds: *information bit* nodes and *frozen bit* nodes. When a frozen bit leaf node is reached, its binary value is unconditionally set to zero. Instead, when an information leaf node is reached, its binary value is set according to the *sign* of its LLR (0 if LLR is positive, 1 otherwise). Once every node in the tree has been visited in both directions, the algorithm eventually updates partial sums in the root node and the decoding process is terminated. If the polar code is not systematic, the decoded bits $\hat{\mathbf{u}}$ are the leaf bits in the tree. Otherwise, if the polar code is systematic, the decoded bits $\hat{\mathbf{u}}$ can be directly extracted from the root node of the polar tree in the form of an N-bit partial sum vectors. In this thesis, only the systematic polar encoding scheme is considered. This construction leads to an improved BER while the decoding computational complexity remains unchanged.

The SC algorithm is a key to construct the polar codes. A density evolution is performed over the SC binary tree to determine the efficient position of the frozen bits. The idea is to construct the polar codes according to the decoder structure. In this manuscript, the Gaussian Approximation (GA) of the density evolution is used [Tri12].

1.3.3.3 Successive Cancellation List Decoding Algorithm

The Successive Cancellation List (SCL) algorithm is an evolution of the SC [TV11]. The SCL algorithm is summarized in Algorithm 1.1. Unlike the SC algorithm, the SCL algorithm builds a list of candidate codewords along the decoding process. At each call of the "updatePaths()" sub-routine (l.16), 2L candidates are generated. A path metric is then evaluated to keep only the L best candidates among the 2L paths. The path metrics are calculated as in [BPB15]. At the end of the decoding process, the candidate codeword with the best path metric is selected in the "selectBestPath()" sub-routine (l.18). The decoding complexity of the SCL algorithm grows as $O(LN \log_2 N)$. This linear complexity in L leads to significant improvements in BER/FER performances compared to the SC decoder, especially for small code lengths.

The authors in [TV11] observed that when a decoding error occurs, the right codeword is often in the final list, but not with the best path metric. They proposed to concatenate a CRC

Algorithm 1.1: SCL decoding algorithm.

```
Data: λ is a 2D buffer ([L][2N]) to store the LLRs.
Data: ŝ is a 2D buffer ([L][N]) to store the bits.
Function SCLDecode(N, o_λ, o_ŝ)        ▷ o_λ and o_ŝ are offsets in λ and ŝ, resp.
    N_{1/2} ← N/2
    if N > 1 then                                                  ▷ not a leaf node
        for p = 0 to L − 1 do                                  ▷ loop over the L paths
            for i = 0 to N_{1/2} − 1 do                          ▷ apply the f function
                λ[p][o_λ + N + i] ← f(λ[p][o_λ + i], λ[p][o_λ + N_{1/2} + i])
        SCLDecode(N_{1/2}, o_λ + N, o_ŝ) ;                  ▷ recursive call to the decoder
        for p = 0 to L − 1 do
            for i = 0 to N_{1/2} − 1 do                          ▷ apply the g function
                λ[p][o_λ + N + i] ← g(λ[p][o_λ + i], λ[p][o_λ + N_{1/2} + i], ŝ[p][o_ŝ + i])
        SCLDecode(N_{1/2}, o_λ + N, o_ŝ + N_{1/2}) ;        ▷ recursive call to the decoder
        for p = 0 to L − 1 do
            for i = 0 to N_{1/2} − 1 do          ▷ update the partial sums (h function)
                ŝ[p][o_ŝ + i] ← h(ŝ[p][o_ŝ + i], ŝ[p][o_ŝ + N_{1/2} + i])
    else                                                              ▷ a leaf node
        updatePaths() ;                          ▷ update, create and delete paths
SCLDecode(N, 0, 0) ;                                          ▷ launch the decoder
selectBestPath()
```

to the codeword in order to discriminate the candidate codewords at the final stage of the SCL decoding. Indeed, this technique drastically improves the FER performance of the decoding process. This algorithm is denoted as the *CRC-Aided SCL* (CA-SCL). In terms of computational complexity, the overhead consists in the computation of L CRCs at the end of each decoding.

1.3.3.4 Simplified Successive Cancellation Class of Algorithms

Frozen bits fully define the decoder leaf values. Hence some parts of the traversal can be cut and its computation avoided, depending on the location of the frozen bits. More generally, the tree functions can be versioned depending on these bits. In [AK11], a tree pruning technique called the Simplified SC (SSC) was applied to SC decoding algorithm. An improved version was proposed in [Sar+14b]. This technique relies on the fact that, depending on the frozen bits location in the leaves of the tree, the definition of dedicated nodes enables to prune the decoding tree: Rate-0 nodes (`R0`) correspond to a sub-tree whose all leaves are frozen bits, Rate-1 nodes (`R1`) correspond to a sub-tree in which all leaves are information bits, REPetition (`REP`) and Single Parity-Check (`SPC`) nodes correspond to repetition and SPC codes sub-trees, respectively. These special nodes, originally defined for SC decoding, can be employed in the case of SCL decoding as long as some modifications are made in the path metric calculation [Sar+16]. This tree-pruned version of the algorithm is called Simplified SCL (SSCL) and CA-SSCL when a CRC is used to discriminate the final candidate codewords. The tree pruning technique can drastically reduce the amount of computations in the decoding process. The Figure 1.9 shows that more than half of the tree nodes can be removed for $N = 8$ and $R = 1/2$ (this is representative of real-life codes).

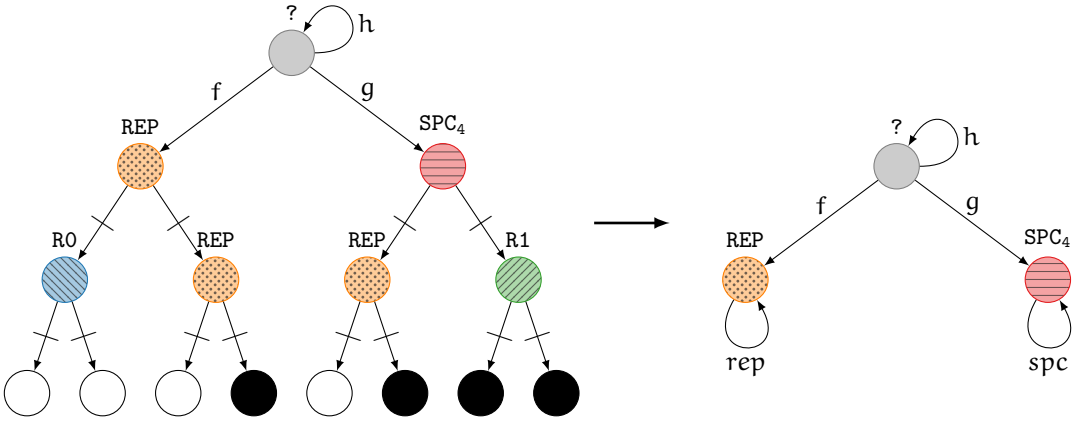


Figure 1.9 – Example of tree pruning on a small binary tree ($N = 8$). The tree is cut and the computations are versioned according to the location of the frozen bits.

1.3.3.5 Adaptive Successive Cancellation List Decoding Algorithm

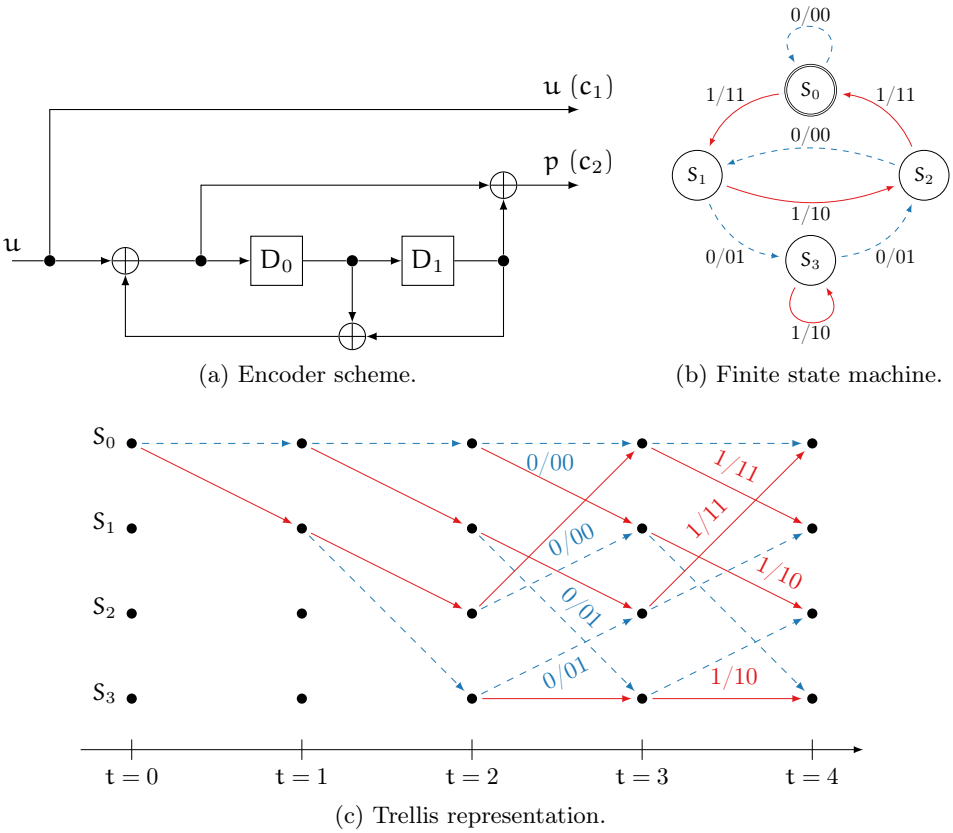
The presence of the CRC can be further used to reduce the decoding time by gradually increasing L. This variation of SCL is called Adaptive SCL (A-SCL) [LST12]. The first step of the A-SCL algorithm is to decode the received frame with the SC algorithm. Then, the decoded polar codeword is checked with a CRC. If the CRC is not valid, the SCL algorithm is applied with $L = 2$. If no candidate in the list satisfies the CRC, L is repeatedly doubled until it reaches the value L_{max}. We call this version of the A-SCL decoding the Fully Adaptive SCL (FA-SCL) as opposed to the Partially Adaptive SCL (PA-SCL), in which the L value is not gradually doubled but directly increased from 1 (SC) to L_{max}. The simplified versions of these algorithms are denoted PA-SSCL and FA-SSCL. In order to simplify the algorithmic range, in the remainder of the manuscript, only the simplified versions are considered. The use of either the FA-SSCL or the PA-SSCL algorithmic improvement introduces no BER or FER performance degradation as long as the CRC length is adapted to the polar code length. If the CRC length is too short, the decoding performance may be degraded because of false detections. These adaptive versions of SSCL can achieve higher throughputs. Indeed, a large proportion of frames can be decoded with a single SC decoding. This is especially true when the SNR is high.

1.3.4 Turbo Codes

1.3.4.1 Coding Scheme

In this sub-section, the convolutional sub-encoder is presented first and then the turbo encoding process is detailed. The first convolutional codes have been introduced by Peter Elias in 1955 [Eli55]. The objective was to propose an alternative to block codes in term of codeword length flexibility: theoretically, the length of a convolutional code is infinite. The coding scheme output depends on the current input and on the few previous inputs.

For a $R = 1/2$ encoder, the current p_k output parity bit can be expressed as a linear combination of the ν previous bits of the message: $p_k = \sum_{j=0}^{\nu} g_j^{(2)} u_{k-j} + \sum_{j=1}^{\nu} g_j^{(1)} p_{k-j}$, where ν represents the number of elements memorized inside the encoder. The sequence of elements g_j is called the code-generating sequence and is often expressed in octal. Figure 1.10a gives three representations of a convolutional code of rate $R = 1/2$ with a memory $\nu = 2$ (D_0 and D_1 are



(a) Encoder scheme.

(b) Finite state machine.

(c) Trellis representation.

Figure 1.10 – Different representations of a recursive and systematic convolutional code ($R = 1/2$).

shift registers). Its two code-generating sequences $\mathbf{g^{(1)}} = (7)_8$ and $\mathbf{g^{(2)}} = (5)_8$ define the $c_2 = p$ output while $c_1 = u$. In the example, the convolutional code has the particularity to be systematic because $c_1 = u$ and recursive because of the feedback loop before the first shift register D_0. In the literature, this type of coding scheme is called Recursive Systematic Convolutional (RSC). Only RSC codes are considered in the document. In Figure 1.10, the number of D memories $\nu = 2$ so, the code can have $2^\nu = 4$ different states. Thus, a convolutional code can be expressed as a finite-state machine as shown in Figure 1.10b. The initial state S_0 corresponds to $D_0 = 0$ and $D_1 = 0$, the state S_1 corresponds to $D_0 = 1$ and $D_1 = 0$, the state S_2 corresponds to $D_0 = 0$ and $D_1 = 1$ and, finally, the state S_3 corresponds to $D_0 = 1$ and $D_1 = 1$. The notation on the edges is in the form of u/c_1c_2. For instance, from the state S_1, if the input bit u is 1, then the encoder will output two bits $c_1 = 1$ and $c_2 = 0$ and will go in the state S_2. This is denoted by *1/10* below the directed edge between S_1 and S_2.

Figure 1.10c introduces another convenient representation of convolutional codes: the trellis. This representation has been used for the first time by Dave Forney in 1973 [For73]. It is especially useful to facilitate the understanding of the decoding process. Indeed, it enables to see the internal state of the encoder, its transitions, and the temporal evolution. However, the purpose of this section is not to detail the decoding process, it will be made in the next section. Considering the encoder initial state S_0, from $t = 0$ the two next possible states are S_0 and S_1. At $t = 1$, the encoder can be in state S_0 or S_1, so the next possible states are S_0, S_1, S_2 or S_3. One can note that starting from $\nu + 1$ time units, the trellis pattern is repeated.

As mentioned before, a turbo code is built from two convolutional codes. Figure 1.11 shows a generic view of the turbo coding process. In the example, the code rate of the turbo code is

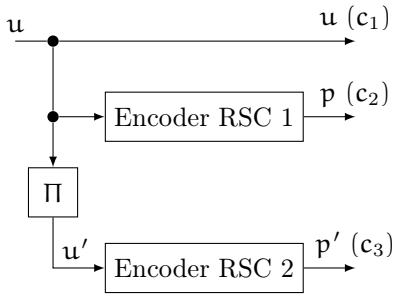


Figure 1.11 – Turbo code ($R = 1/3$) with two convolutional sub-encoders and a Π interleaver.

$R = 1/3$. This rate is obtained from two convolutional sub-encoders of rate $R = 1/2$ (like the one shown in Figure 1.10). The two parity bits p and p' are obtained from the c_2 outputs of the convolutional codes while the systematic c_1 outputs are ignored. The first sub-encoder encodes the u input bit while the second one encode the u' bit. The u' bit is determined from u after a Π interleaving process. For each u_k bit there is a single u'_k associated bit in the K input information bits. The interleaving process is a key point for the efficiency of a turbo code. The interleaving process permutes the information bits from their *natural (sequential) order* into the *interleaved order*. The permutation function defines the interleaver type. The K information bits in the natural order are given to the sub-encoder 1 while the K information bits in the interleaved order are given to the sub-encoder 2.

1.3.4.2 Turbo Decoding Algorithm

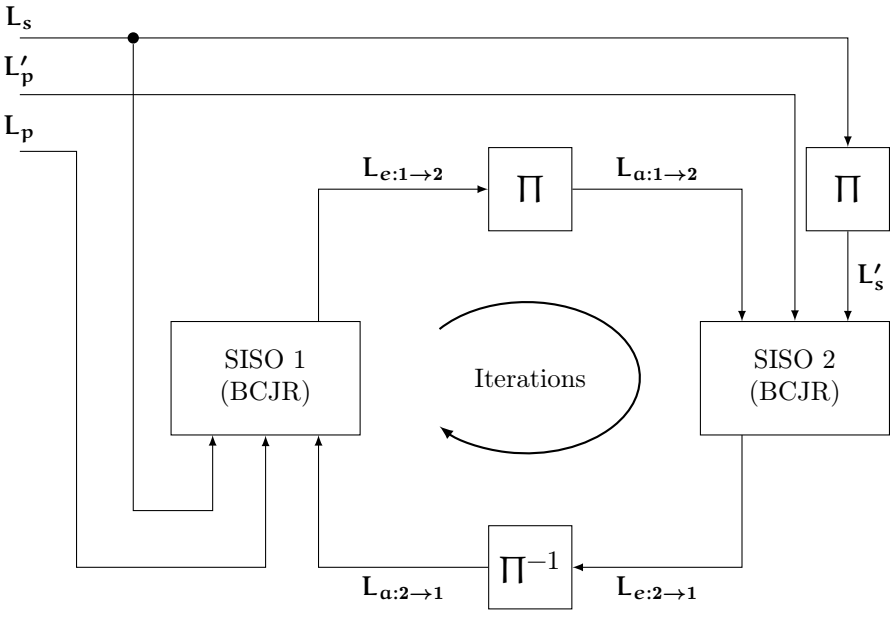


Figure 1.12 – Information exchanges in turbo decoding process.

The turbo decoder consists of two component decoders exchanging soft information in terms of log-likelihood ratio (LLR) for each transmitted information bit through an interleaver and a deinterleaver. Figure 1.12 illustrates the internal structure of a turbo decoder. Two Soft Input Soft Output (SISO) decoders are represented with the interleaving process Π and the deinterleaving process Π^{-1}. In our work, only rate $R = 1/3$ codewords are considered. K represents the number of information bits and N is the codeword size: $N = K \times 3$.

Algorithm Outline Turbo decoding is carried out over several iterations. Each iteration consists of two component decoding phases. During each phase, a component decoder performs a maximum a posteriori (MAP) decoding based on the BCJR algorithm [Bah+74], which generates so-called extrinsic LLRs given the LLRs obtained by the detector and a priori LLRs obtained from the other component decoder. The BCJR algorithm consists of one forward and one backward traversal on a trellis, which are defined by the underlying code. Specifically, to decode a codeword of K information bits, the BCJR algorithm performs the following steps: (i) the branches of the trellis are weighted from the systematic LLRs ($\mathbf{L_s}$), the parity LLRs ($\mathbf{L_p}$) and the a priori information ($\mathbf{L_a}$); (ii) in the forward traversal step, it computes K sets of forward state metrics for each transmitted information bit; (iii) in the backward traversal step, it computes K sets of backward state metrics for each transmitted information bit; (iv) to compute the extrinsic LLRs ($\mathbf{L_e}$), the BCJR algorithm then combines the forward and backward state metrics. The $\mathbf{L_s}$, $\mathbf{L_p}$, $\mathbf{L'_p}$ vectors of LLRs correspond to the decoder input $\mathbf{l}$ split into 3 sub-sets.

Algorithm 1.2: Pseudo-code of the BCJR decoding algorithm.

for $k = 0;\ k < K;\ k = k + 1$ **do** ▷ (i) parallel loop
  $\boldsymbol{\gamma}^k \leftarrow \text{computeGamma}(L_s^k, L_p^k, L_a^k)$
$\boldsymbol{\alpha}^0 \leftarrow \text{initAlpha}()$
for $k = 1;\ k < K;\ k = k + 1$ **do** ▷ (ii) sequential loop
  $\boldsymbol{\alpha}^k \leftarrow \text{computeAlpha}(\boldsymbol{\alpha}^{k-1}, \boldsymbol{\gamma}^{k-1})$
$\boldsymbol{\beta}^{K-1} \leftarrow \text{initBeta}()$
for $k = K - 2;\ k \geqslant 0;\ k = k - 1$ **do** ▷ (iii) sequential loop
  $\boldsymbol{\beta}^k \leftarrow \text{computeBeta}(\boldsymbol{\beta}^{k+1}, \boldsymbol{\gamma}^k)$
for $k = 0;\ k < K;\ k = k + 1$ **do** ▷ (iv) parallel loop
  $L_e^k \leftarrow \text{computeExtrinsic}(\boldsymbol{\alpha}^k, \boldsymbol{\beta}^k, \boldsymbol{\gamma}^k, L_s^k, L_a^k)$

Algorithm 1.2 summarizes the previously enumerated steps in a pseudo-code. $\boldsymbol{\gamma}$ are the values of the trellis branches, $\boldsymbol{\alpha}$ are the values of the nodes in the forward traversal of the trellis and $\boldsymbol{\beta}$ are the values of the nodes in the backward traversal of the trellis.

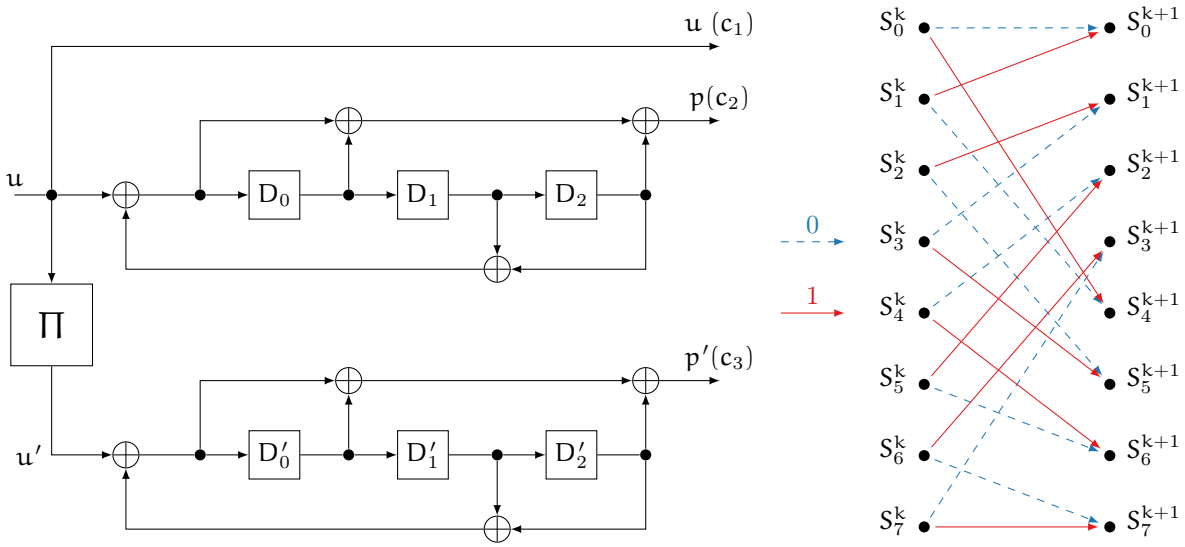


Figure 1.13 – Turbo LTE encoder and its associated 8-state trellis. $\mathbf{g^{(1)}} = (13)_8$, $\mathbf{g^{(2)}} = (15)_8$.

In this thesis, we focus on the turbo codes of the LTE standard [ETS13] (3G and 4G mobile

networks). Figure 1.13 gives the definition of one LTE turbo encoder. This encoder leads to an 8-state trellis. In the next sections and chapters, this LTE trellis is always considered.

Branch-metric Computations Let S_j^{k+1} be the j^{th} state associated with information bit $k+1$ and $j \in \{0, 7\}$. There are two incoming branches into state S_j^{k+1}. Each incoming branch is associated with values u^k and p^k, the k^{th} information bit and the parity bit (both ± 1), respectively. The branch metrics associated with states S_i^k and S_j^{k+1} are computed as follows:

$$\gamma(S_i^k, S_j^{k+1}) = 0.5(L_s^k + L_a^k)u^k + 0.5(L_p^k p^k). \tag{1.8}$$

L_s^k and L_a^k are the systematic channel LLR and the a priori LLR for k^{th} trellis step, respectively. In the BCJR SISO decoder 1, the $\mathbf{L_s}$, $\mathbf{L_p}$ and $\mathbf{L_a}$ vectors of LLRs are considered in the natural domain while in the BCJR SISO decoder 2, the $\mathbf{L'_s}$, $\mathbf{L'_p}$ and $\mathbf{L'_a}$ LLRs are used instead in the interleaved domain. However, the computations in the natural and in the interleaved domain are similar. That is the reason why only the operations in the natural domain are described here. Note that it is not necessary to evaluate the branch metric $\gamma(s^k, s^{k+1})$ for all 16 possible branches, as there are only four different branch metrics: $\gamma_0^k = 0.5(L_s^k + L_a^k + L_p^k)$, $\gamma_1^k = 0.5(L_s^k + L_a^k - L_p^k)$, $-\gamma_0^k$, and $-\gamma_1^k$.

Forward and Backward State-metric Computations The forward state metrics have to be computed recursively from trellis step to trellis step. The forward state metrics of step $k+1$ correspond to the vector $\boldsymbol{\alpha^{k+1}} = [\alpha_0^{k+1}, ..., \alpha_7^{k+1}]$, where the j^{th} forward state metric α_j^{k+1} only depends on two forward state metrics of stage k. These state metrics are computed as:

$$\alpha_j^{k+1} = \max_{i \epsilon F}{}^*(\alpha_i^k + \gamma(S_i^k, S_j^{k+1})), \tag{1.9}$$

where the set F contains the two indexes of the states in step k connected to state S_j^{k+1} (as defined by the trellis). The max* operator can be expressed as follow:

$$\max^*(a, b) = \max(a, b) + \log(1 + \exp(-|a - b|)), \tag{1.10}$$

where $\log(1 + \exp(-|a - b|))$ is a correction term. Computations of the backward state metrics are similar to that of the forward trellis traversal in Equation 1.9. The vector of backward state metrics, denoted by $\boldsymbol{\beta^k} = [\beta_0^k, ..., \beta_7^k]$, is computed as:

$$\beta_j^k = \max_{i \epsilon B}{}^*(\beta_i^{k+1} + \gamma(S_j^k, S_i^{k+1})), \tag{1.11}$$

where B is the set containing the indexes of states in step $k+1$ connected to state S_j^k as defined by the trellis.

Extrinsic LLR Computations After the forward and backward recursions have been carried out, the extrinsic LLRs for the k^{th} bit are computed as follow:

$$\begin{aligned} L_e^k = &\max_{\{S_k, S_{k+1}\} \epsilon U^1}{}^* \left(\alpha_i^k + \beta_j^{k+1} + \gamma(S_i^k, S_j^{k+1})\right) \\ &- \max_{\{S_k, S_{k+1}\} \epsilon U^{-1}}{}^* \left(\alpha_i^k + \beta_j^{k+1} + \gamma(S_i^k, S_j^{k+1})\right) \\ &- L_s^k - L_a^k, \end{aligned} \tag{1.12}$$

where the sets $\mathcal{U}^1$ and $\mathcal{U}^{-1}$ designate the set of states connected by paths where $u^k = 1$ and the set of states connected by paths where $u^k = -1$ (BPSK mapping), respectively.

Approximation of the MAP Operations in the BCJR Decoder The max* operator of the MAP algorithm is compute intensive, mainly due to the logarithm and exponential functions in the correction term. It can be approximated as follow: $\max^*(a, b) \approx \max(a, b)$. The correction term is simply removed. This approximation is called the *max-log-MAP* algorithm (ML-MAP). Its low computational complexity makes efficient software and hardware implementations possible. However, the ML-MAP algorithm can negatively affect the decoding performance. Yet, it is possible to partially recover the error-rate performance loss by scaling the extrinsic LLRs in the turbo decoder by a factor α. This version is called the *Enhanced* max-log-MAP (EML-MAP) algorithm [VF00, Stu+11].

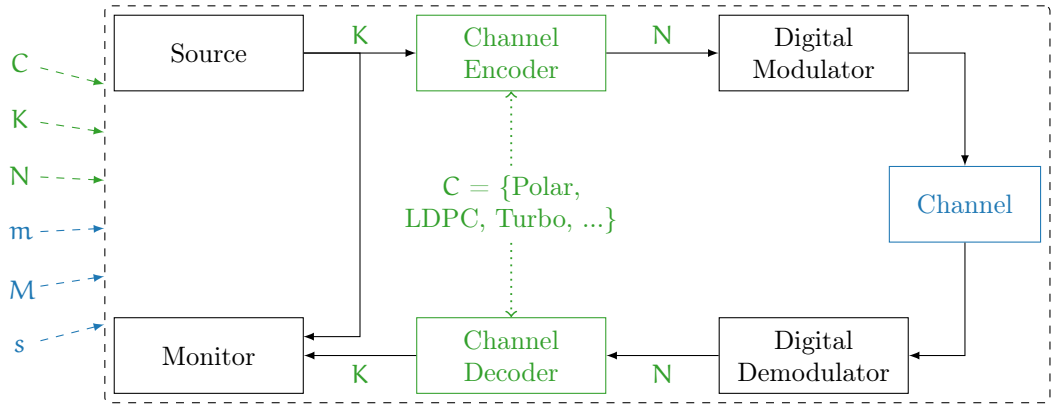
1.4 Applicative Contexts

In the previous section, three channel code families have been presented. In this thesis we focus on software implementations of the previously introduced channel decoders. These decoders can be used in different applicative contexts such as simulations or in real-time systems for instance. This section describes the different applicative contexts that will be considered all along the manuscript.

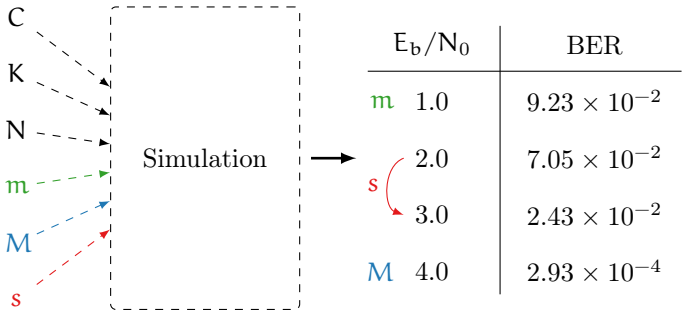
1.4.1 Functional Simulation

There are many possible coding schemes with different characteristics. The previous section introduced several families of FEC codes that are present in most of common standards. Before they were standardized, these codes have been evaluated and compared. The codes can be applied in a infinity of different ways. For each coding scheme, many different decoding algorithms can be implemented. It is then mandatory to be able to evaluate and compare the BER/FER decoding performance of selected decoders with each others before implementing them in real systems. To this purpose, the channel coding designers use functional simulation. On an AWGN channel, depending on the channel code, it is possible to predict with varying degrees of computational effort the BER/FER decoding performance of a digital communication system. For simple channel codes like BCH, RS and convolutional codes it is possible to analytically evaluate the performance while for more complex codes like LDPC, turbo and polar codes it becomes more difficult to use direct methods. The solution is then to resort to a compute intensive Monte Carlo simulation of the digital communication system. The idea is to evaluate the performance of the system by generating many random frames and applying random noise samples on these frames, the noisy frames are decoded and the output sequence of bit $\hat{\mathbf{u}}$ is compared with the initial information bits $\mathbf{u}$. The error count is then used to update the BER/FER value until they reach a stable value.

Figure 1.14a describes a simulation sequence similar to the digital communication chain presented in Figure 1.1. The only difference is that the *sink* block has been replaced by what we call a *monitor*. The *monitor*, unlike the *sink*, knows the K output information bits from the *source* ($\mathbf{u}$). The C, K and N parameters define the type of code/decoder, the number of information bits and the codeword size, respectively. These parameters have a direct impact on the selection of the *channel encoder* and the *channel decoder* blocks in the simulation. Figure 1.14b shows the BER output of the resulting simulation. The m, M and s parameters enable to control the



(a) Specification of the simulation chain.

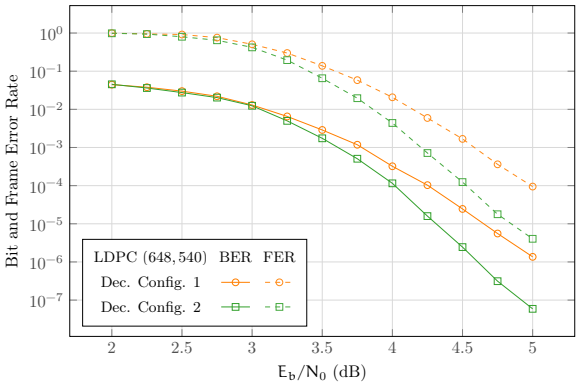


(b) Input simulation parameters and output BER results.

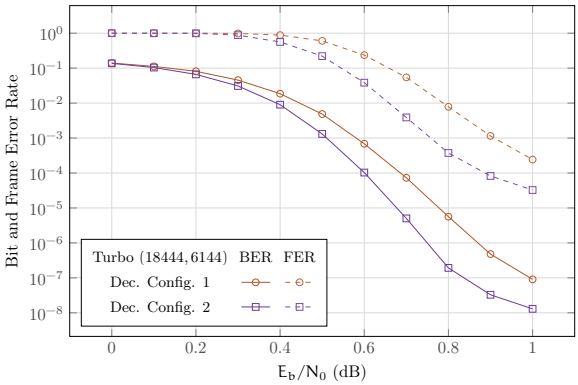
Figure 1.14 – Description of a digital communication system simulation.

AWGN channel noise. m is the minimum SNR value to simulate in the channel, while M is the maximum SNR value. s is the SNR step between two SNR values.

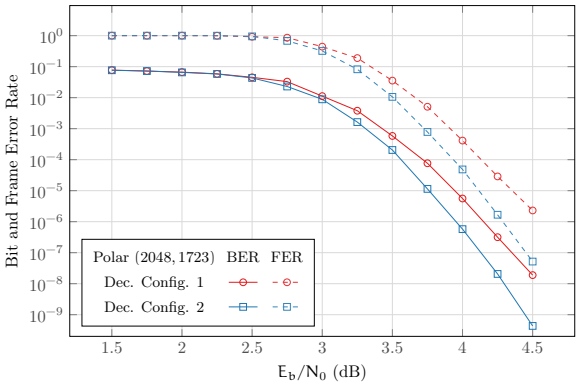
As an illustration, Figure 1.15 presents BER and FER decoding performances estimated with Monte Carlo simulations on a large variety of code families and decoding parameters. On each graphic, the BER is plotted with solid lines while the FER is plotted with dashed lines. Both the BER and the FER depend on the SNR ($\mathsf{E_b/N_0}$). The higher the SNR is, the lower the noise is and therefore the number of errors. In Figure 1.15a, for instance, on the same LDPC code ($\mathsf{N} = 648$ and $\mathsf{K} = 540$) and considering a 5.0 dB SNR, the configuration 2 of the decoder achieves better decoding performance than the configuration 1 because the green curve is below the orange curve. In other terms, lower is better. When multiple configurations are shown together, they are ordered by increasing BER/FER performance. This is achieved at the cost of a higher computational effort during the decoding process compared to the first configuration. The purpose of Figure 1.15 is not to compare codes with each others but to introduce the typical BER/FER curves that will be used in the next chapters. It shows that there is a large set of possible combinations of codes and decoding configurations.



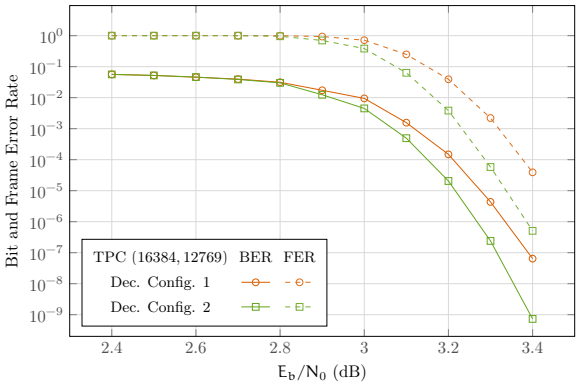
(a) LDPC code ($R = 5/6$).



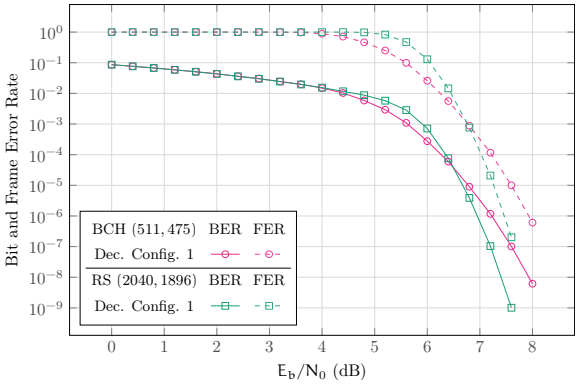
(b) Turbo code ($R \approx 1/3$).



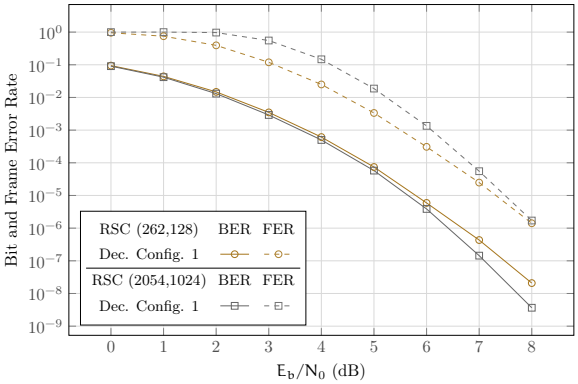
(c) Polar code ($R \approx 0.84$).



(d) Turbo product code ($R \approx 0.78$).



(e) Algebraic codes ($R \approx 0.93$).



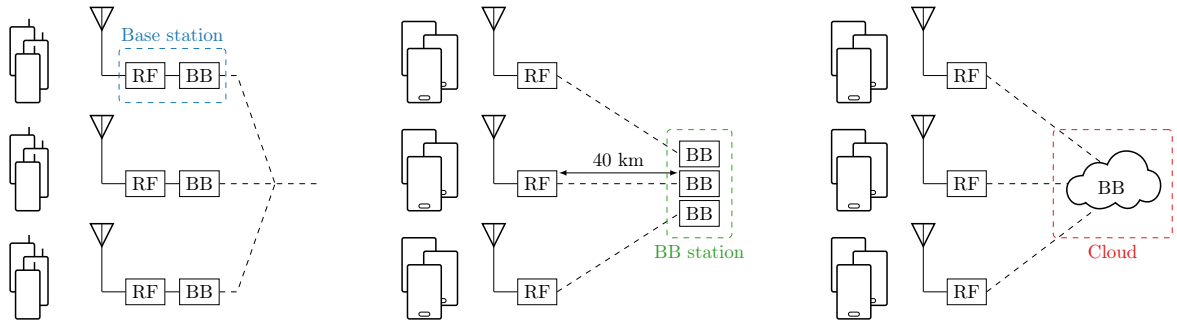
(f) Convolutional codes ($R \approx 1/2$).

Figure 1.15 – BER and FER simulation results on various code families and decoder configurations. Lower is better. The codes are given in the (N, K) form.

1.4.2 Software-defined Radio

A Software-Defined Radio (SDR) is a radio communication system where components traditionally implemented in hardware are instead implemented by means of software. The concept was first introduced by Joseph Mitola in 1992 [Mit92, Mit93]. Since then, SDR systems have been used in various contexts such as military needs or amateur radio for instance. Recently, the SDR is considered a good candidate for the 5G wireless mobile network and more generally in cloud radio access networks. This is detailed in the next paragraphs.

A cellular network or mobile network is a communication network where the last link is wireless. The principle of mobile networks is to divide the territory into zones called "cells". Each cell is associated to a base station and a number of frequency channels to communicate with mobile terminals. Each base station is connected to the networks handling voice calls, text messages and data transfers. As standards evolve, the structure of mobile networks changes in order to increase throughput and latency performance and also the number of connected terminals. The objective is to cope with the exponential growth of terminals.



(a) Early base station. (b) BB and RF processing separation. (c) C-RAN.

Figure 1.16 – Base stations evolution in mobile networks.

In the two first generations of mobile networks (1G and 2G), all the signal processing was treated in the base station near the antenna (see Figure 1.16a). Since the third generation of mobile networks (3G) two types of processing have been separated: 1) the radio frequency (RF) processing on the one side and 2) the base band (BB) processing on the other side. The RF processing is attached to the antenna while the BB processing is shared among multiple antennas (see Figure 1.16b). The range of this type of station is approximately 40 kilometers. The connexions between the antennas and the BB station are made through wired links. The RF processing mainly converts the analog signal into a digital one (or the other way around) while the BB station performs all the digital processing including the encoding/decoding and the digital modulation/demodulation. The purpose of separation of the RF and BB processing is to be able to put the BB stations near the urban centers and so to reduce their cost of maintenance.

The virtualization of the mobile network is considered by industrial actors [Hua13, Eri15] and academic ones [Wub+14, Ros+14, Che+15] as a promising evolution. This is also known as the *Cloud Radio Access Network* (C-RAN): it is proposed for a part of the processing traditionally made in the base stations. In this network structure, the computational hardware resources of the BB processing are shared between multiple antennas (see Figure 1.16c). This enables new optimizations: 1) better adaptation to non-uniform traffic; 2) energy saving; 3) higher throughput and lower latency; 4) scalability and maintainability increase [Che+15]. Thus, the computational BB units have to be virtualized: there should no longer be hardware dedicated to specific antenna. Contrariwise, the BB computational effort has to be distributed at the cloud level.

From the first to the fourth generation of mobile networks, the BB processing was systematically made on Application-Specific Integrated Circuits (ASICs or dedicated hardware). However, with the emergence of the C-RAN more flexible solutions like the software ones are seriously considered. On the receiver side, the algorithms can be compute intensive (especially the digital demodulation and the channel decoding). Knowing that, a main challenge is to be able to achieve high throughput and low latency software implementations as well as flexible ones [Nik15, RG17].

1.4.3 Sparse Code Multiple Access

Non-Orthogonal Multiple Access (NOMA) mechanisms are investigated as means to improve the fifth-generation mobile communication systems (5G) [Isl+17] to support massive connectivity and to reduce bit error rates. Sparse Code Multiple Access (SCMA) is a NOMA mechanism that offers better bit error rate performance and higher spectral efficiency, while the sparsity of the codebooks ensures a lower complexity of decoding compared to other non-orthogonal modulations [NB13]. SCMA is a promising candidate for 5G communication systems since it provides up to 3 times more connectivity by spreading the information of each user's codebook over sets of shared Orthogonal Frequency-Division Multiplexing (OFDM) [Alt15a]. According to the NGMN white paper [NGM15], 5G targets more diverse scenarios compared to 4G. Applications can be broadband support in dense areas, low latency connectivity for Augmented Reality (AR) and reliable communication for intelligent industrial controls, Internet of Things (IoT) or Internet of Mission Critical Things (IoMCT). Unfortunately, the massive connectivity and spectral efficiency of SCMA come at the cost of high complexity in the decoder, making the design of high throughput and low complexity decoders a challenge [Lu+15]. In this thesis, we propose to study the SCMA system as it is usually combined with the channel code families presented before. It is introduced in this section and revised later in light of the needs of Cloud Radio Access Networks (C-RANs).

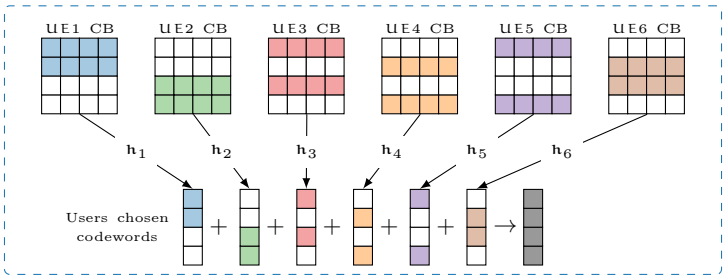
1.4.3.1 Overview of the System Model

In this section, scalar, vector and matrix are presented as x, $\mathbf{x}$, $\mathbf{X}$ respectively. The n^{th} element of a vector denoted by $\mathbf{x}_n$ and $\mathbf{X}_{n,m}$ is the element of n^{th} row and m^{th} column of matrix $\mathbf{X}$. Notation diag(x) shows a diagonal matrix where its n'th diagonal element is $\mathbf{x_n}$. In addition, the transpose of a matrix is expressed as $\mathbf{X}^{\mathsf{T}}$.

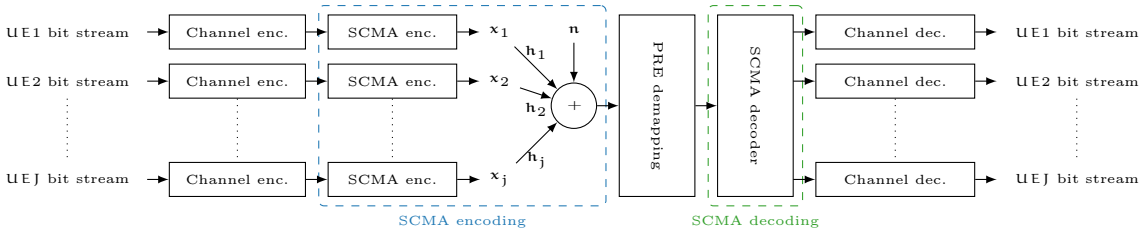
An SCMA encoder with J users (layers) and K physical resources is a function that maps a binary stream of data to K-dimensional complex constellations $f : \mathbb{B}^{\log_2(M)} \rightarrow \mathbb{X}, x = f(\mathbf{b})$ where $\mathbf{X} \subset \mathbb{C}^k$. The K-dimensional complex codeword x is a sparse vector with $N < K$ non-zero entries. Each layer $j = 1, ..., J$ has its own codebook to generate the desired codeword according to the binary input stream. Figure 1.17 shows the SCMA uplink chain with $J = 6$, $K = 4$ and $N = 2$. SCMA codewords are spread over K physical resources, such as OFDM tones. Figure 1.17a shows that in the multiplexed scheme of SCMA, all chosen codewords of the J layers are added together after being multiplied by the channel coefficient $\mathbf{h}_j$. Then, the entire uplink chain is shown in Figure 1.17b. The output of the SCMA encoder is altered by a white additive noise $\mathbf{n}$:

$$\mathbf{y} = \sum_{j=1}^{J} \text{diag}(\mathbf{h}_j)\mathbf{x}_j + \mathbf{n}, \tag{1.13}$$

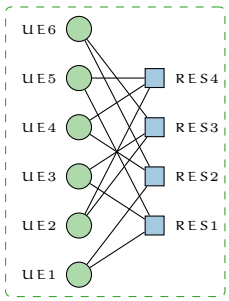
where $\mathbf{x}_j = (x_1, ..., x_{Kj})^{\mathsf{T}}$ and $\mathbf{h}_j = (h_1, ..., h_{Kj})^{\mathsf{T}}$ are respectively codeword and channel coefficients



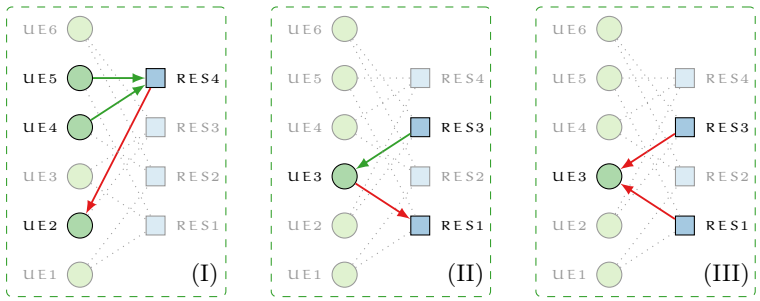
(a) SCMA encoder with 6 users (layers) and 4 physical resources.



(b) SCMA uplink chain with channel coding.



(c) Factor graph representation of a decoder.



(d) Message Passing Algorithm based on Bayesian factor graph: (I) Resource to user message, (II) Guess swap at each user and user to resource message, (III) Final guess at each user.

Figure 1.17 – SCMA system model, encoding and decoding schemes.

of layer j. Considering the digital communication chain presented in Figure 1.1 the SCMA encoder can be seen as a digital modulator and the SCMA decoder can be seen as a digital demodulator.

1.4.3.2 Message Passing Algorithm Decoding Scheme

Exploiting sparsity of the codebooks, Message Passing Algorithm (MPA) decoders were introduced to achieve very good decoding performance with lower complexity [Zha+14a]. Figure 1.17c shows a Bayesian factor graph representation of an MPA decoder with six users and four physical resources. Thanks to the sparsity of the codebooks, exactly three users collide in each physical resource. There are four possible codewords for each of the three connected user's codebooks, which gives 64 possible combined codewords in each physical resource.

In the first step of the MPA, the 64 distances between each possible combined codeword and the actual received codeword are calculated:

$$d_{RES\beta}(\mathbf{m}, \mathbf{H}) = \|\mathbf{y}_\beta - \sum_{l \subset \zeta, m_u \in \{1,\dots,K\}} \mathbf{h}_{l,m_u} \mathbf{x}_{l,m_u}\|, \tag{1.14}$$

where ζ is the set of users connected to resource β and the considered codeword is denoted

as m. Assuming perfect channel estimation and Gaussian noise, these Euclidean distances can be expressed as probabilities using (1.15):

$$\Psi(d_{RES\beta}) = \exp\left(-\frac{d^2_{RES\beta}}{2\sigma^2}\right). \tag{1.15}$$

After calculating the residual probability of each codeword with (1.15), iterative MPA starts exchanging beliefs on possible received codewords among the users and resources nodes of the factor-graph. According to Figure 1.17d (I), a message from resources to users has been defined to contain extrinsic information of two other connected users. For instance, a message from resource 4 to user 2 containing the probability information of codeword i can be expressed as:

$$\mu_{RES4\to UE2}(i) = \sum_{j=1}^{4}\sum_{i=1}^{4}\Psi\Big(d_{RES4}(i,j,k,\mathbf{H})\Big) \times \mu_{UE4\to RES4}(j) \times \mu_{UE5\to RES4}(k). \tag{1.16}$$

As shown in Figure 1.17d(II) there are only two resources connected to each user. A message from a user to a resource is a normalized guess swap at the user node:

$$\mu_{UE3\to RES1}(i) = \frac{\mu_{RES3\to UE3}(i)}{\sum_i \mu_{RES3\to UE3}(i)}. \tag{1.17}$$

Message passing between users and resources (see (1.16) and (1.17)) will be repeated three to eight times to reach the desired decoding performance. The final belief at each user $B(i)$ is the multiplication of all incoming messages as illustrated in Figure 1.17d(III) and (1.18) for UE3 and codeword i. Finally, (1.19) is used to calculate soft outputs for $\hat{\mathbf{x}}$:

$$B_3(i) = \mu_{RES1\to UE3}(i) \times \mu_{RES3\to UE3}(i), \tag{1.18}$$

$$\hat{\mathbf{x}} = \ln\left(\frac{P(\mathbf{y}|b_x=0)}{P(\mathbf{y}|b_x=1)}\right) = \ln\left(\frac{\sum_m B_m(i) \text{ when } b_x=0}{\sum_m B_m(i) \text{ when } b_x=1}\right). \tag{1.19}$$

Log-MPA Since calculation of exponentials in (1.15) requires relatively high computational effort, changing the algorithm to logarithmic domain simplifies (1.16) in:

$$\mu_{RES1\to UE5}(i) = \max_{j,k=1,\dots,4}\left(-\frac{d^2_{RES1}(i,j,k,\mathbf{H})}{2\sigma^2}\right) + \mu_{UE2\to RES1}(j) + \mu_{UE3\to RES1}(k), \tag{1.20}$$

due to elimination of exponential's high dynamic ranges, there is no need to normalize the guess swap and $\mu_{UE3\to RES1}(i) = \mu_{RES3\to UE3}(i)$. The rest of the algorithm can be expressed as follows:

$$B_3(i) = \mu_{RES3\to UE3}(i) + \mu_{RES1\to UE3}(i), \tag{1.21}$$

$$\hat{\mathbf{x}} = \max_i(B_m(i)) \text{ when } b_x=0 - \max_i(B_m(i)) \text{ when } b_x=1. \tag{1.22}$$

Estimated-MPA (E-MPA) Computation of the exponentials in (1.15) is one of the most important bottlenecks of the MPA algorithm. It is possible to further accelerate the computation by using proper estimations. The exact exponential computation is not essential to produce a satisfying estimation in the MPA algorithms. Considering that (1.15) represents a Gaussian PDF, it can be replaced by sub-optimal bell-shaped polynomial distributions to model the noise. It will be shown in Section 4.5.2 that using a polynomial estimation can increase the throughput while leading to marginal bit error rate degradation after the MPA decoding. However, these estimated probabilities cause small degradations of the FER performance after the channel decoding. The proposed PDF must satisfy two conditions to be valid: 1) it must be positive and lower bounded at zero, 2) its integral over $(-\infty, \infty)$ must be equal to 1. The following function is suggested to estimate the exponentials:

$$\Psi'_{d_{RES\beta}} = \frac{2/\pi}{2\sigma^2 + 4d^4_{RES\beta}}. \tag{1.23}$$

The computation of Ψ' is faster than the original Ψ [Gha+17, 2]. The probabilities produced using (1.15) and (1.23) are normalized according to (1.17). Furthermore, the numerator $2/\pi$ does not play an important role in MPA and can be uniformly eliminated from all calculations to reduce the computational effort. Thus,

$$\Psi'_{d_{RES\beta}} \approx \frac{1}{2\sigma^2 + 4d^4_{RES\beta}}, \tag{1.24}$$

can be used as a systematic replacement to the exponential calculations.

1.5 Problematics

On the eve of the 5G mobile communication generation, the challenge is now to design communication systems able to transmit huge amounts of data in a short time, at a small energy cost, in a wide variety of environments. Researchers work at refining existing coding schemes further, to get low residual error rates with fast, flexible, low complexity decoding processes.

Functional Simulation The validation of a coding scheme requires to estimate its error rate performance. Usually, no simple mathematical model exists to predict such performance. The only practical solution is to perform a Monte Carlo simulation of the whole chain. It means that some data are randomly generated, encoded, modulated, noised, decoded, and the performance is then estimated by measuring the Bit Error Rate (BER) and the Frame Error Rate (FER) at the sink. This process has the advantage of being universal but it also leads to three main problems:

1. **Simulation time:** ~ 100 erroneous frames have to be simulated to accurately estimate the FER/BER. Thus, measuring a FER of 10^{-7} requires the simulation of the transmission of $\sim 100 \times 10^7 = 10^9$ frames. Assuming a frame of 1000 bits, the simulator must then emulate the transmission of 10^{12} bits. Keeping in mind that the decoding algorithm computational complexity may be significant, several weeks or months may be required to accurately estimate the FER/BER of a coding scheme (especially at low error rates).
2. **Algorithmic heterogeneity:** A large number of channel codes have been designed over time. For each kind of code, several decoding configurations are possible. While it is straightforward to describe a unique coding scheme, it is more challenging to have a unified

software description that supports all the coding schemes and their associated algorithms. This difficulty comes from the heterogeneity of the data structure necessary to describe a channel code and the associated decoder: turbo codes are based on trellis schemes, LDPC codes are well-defined on factor graphs and polar codes are efficiently decoded using binary trees.

3. **Reproducibility:** It is usually tedious to reproduce results from the literature. This can be explained by the large amount of empirical parameters necessary to define one communication system, and the fact that not all of them are always reported in publications. Moreover, the simulator source codes are rarely publicly available. Consequently, a large amount of time is spent "reinventing the wheel" just to be able to compare to the state-of-the-art results.

Software-defined Radio Moreover, the Software-Defined Radio (SDR) paradigm is now considered in real communication systems. To match the real-time constraints, here are the main challenges:

1. **High throughput:** New applications, like the video streaming, can be very data-intensive. As a consequence, the compute intensive blocks of the transceiver have to be well optimized to reach levels of performance comparable with the hardware implementations.
2. **Low latency:** Reaching high throughput is not always the major constraint, for instance, in audio-conferencing applications it is uncomfortable to perceive a delay when people are speaking.
3. **Flexibility:** The software implementations have to be able to adapt to various configurations. For instance, when the SNR is changing, the code rate `R` of the decoder can be switched on the fly.
4. **Portability:** The proposed solutions can be deployed on high-end servers as well as on embedded low power systems. Moreover, many operating systems coexist, and it is important to be able to support the most famous ones like Windows, macOS and Linux.

2 Optimization Strategies

This chapter focuses on optimization strategies dedicated to digital communication algorithms. Our contributions are split in two parts: 1) generic strategies and 2) specific optimizations. The two first sections describe the generic strategies we proposed to optimize the algorithms of digital communication receivers. Vectorization is a key of efficient software implementations. A specific wrapper library as well as a sub-set of generic parallelism levels are proposed. The sections afterward are dedicated to the efficient software implementation of the algorithms for digital communication receivers presented in the Chapter 1. In the last section, the proposed contributions are summarized and discussed.

2.1 MIPP: A C++ Wrapper for SIMD Instructions

Recent articles have proposed several optimized software decoders, corresponding to different channel codes: LDPC codes [LJ15, LJ16, LLJ18], polar codes [Gia+16, Sar+16, 4, 5, 3], turbo codes [Zha+12, Wu+13, 6, LJ19]. All of these works show the possibility to reach a good level of *computing performance* by making extensive use of SIMD (Single Instruction Multiple Data) units. This is often achieved at the price of a reduced *flexibility*, by resorting to specific intrinsics, or by making assumptions on the data types. However, these decoders should be implemented in a single source code, in which the following parameters could be changed at runtime: the channel code type, the decoding algorithm, the number of decoding iterations, the data format, etc. Another important aspect is the *portability* of the source code on different processors (Intel® x86, Xeon Phi™ KNL and ARM®) and the possibility to use different instruction sets (SSE, AVX, AVX-512, NEON). These three constraints (performance, flexibility, portability) push towards the use of a SIMD library that helps in the abstraction of the SIMD instruction sets, while still allowing a fine grain tuning of performance.

As a foundation of the thesis work, we propose a new C++ SIMD library, covering the needs in terms of expressiveness and of performance for the algorithms dedicated to channel codes. Our contributions are:

— A portable and high performance C++ SIMD library called MIPP, for SSE, AVX, AVX-512 and NEON instruction sets;

— A comparison with other state-of-the-art SIMD libraries on a Mandelbrot code, demonstrating that the code based on MIPP has similar performance as hand-written intrinsics.

In order to let the compiler inline library calls, which is critical for the intended SIMD programming model purpose, such library are usually header-only. Thus, we refer to them as *wrappers* instead of *libraries*. The MIPP programming model is built on top of intrinsics, enabling a good control on performance, but still provides an abstraction on the basic types used in vectors (ranging from double to byte) and complex operations (parametric reductions, log, exponential, ...).

The MyIntrinsics++ library (MIPP) is a portable wrapper for SIMD intrinsics written in the C++ language. It relies on C++ compile-time template specialization techniques to replace supported generic functions with inline calls to their intrinsics counterpart, for a given instruction set. While MIPP is mostly written in C++98, it requires a C++11-compliant compiler due to the use of convenient features such as the `auto` and `using` keywords. MIPP is open-source (under the MIT license) and available on GitHub [1].

MIPP provides two application programming interface levels. The *Low Level Interface* (low) implements a basic thin abstraction layer directly on top of the intrinsics. The *medium level interface* (med.), built on top of MIPP low, abstracts away more details to lessen the effort from the application programmer by relying on object encapsulation and operator overloading.

2.1.1 Low Level Interface

MIPP low is built around a unified `mipp::reg` type that abstracts vector registers. The vector register type represents hardware registers independently of the data type of the vector elements. MIPP uses the longest native vector length available on the architecture. This design choice preserves programmer flexibility, for instance in situations such as mixing fixed-point and floating-point operations. MIPP also defines a *mask* register type `mipp::msk`, which either

1. MIPP repository: https://github.com/aff3ct/MIPP

directly maps to real hardware masks on instruction sets that support it (such as AVX-512), or to simple vector registers otherwise.

MIPP low defines a set of functions working with `mipp::reg` and `mipp::msk`. This set is organized into eight families: memory accesses, shuffles, bitwise boolean arithmetic, integer operations, float. operations, mathematical functions, reductions, and mask operations.

In the AVX-512 instruction set, one *regular* vector operation plus one masking operation can be performed in a single CPU clock cycle. For instance, the following instruction `"m ? a+b : src"` performs an addition and a masking operation:

```
__m512 _mm512_mask_add_ps(__m512 src, __mmask16 m, __m512 a, __m512 b);
```

MIPP natively supports such operations with the `mipp::mask` function. The previous example becomes in MIPP:

```
mipp::mask<float,mipp::add<float>>(m, src, a, b);
```

For instruction sets without masking support, the `mipp::mask` call is expanded as an operation and a blend instead.

2.1.2 Medium Level Interface

```
template <typename T>
class Reg
{
    // the register type from the MIPP low interface
    mipp::reg r;
    // a simple class constructor encapsulates the load instruction
    Reg(const T *ptr) : r(mipp::load<T>(ptr)) {}
    // the definition of the 'add' method
    inline Reg<T> add(const Reg<T> r) const {
        return mipp::add<T>(r,r.r);
    }
    // overriding of the '+' operator using the previously defined 'add' method
    inline Reg<T> operator+(const Reg<T> r) const {
        return this->add(r);
    }
    /* ... */
};
```

Listing 2.1 – Medium level interface encapsulation.

The MIPP medium level interface (MIPP med.) provides additional expressiveness to the programmer. `mipp::reg` and `mipp::msk` basic types are encapsulated in `mipp::Reg<T>` and `mipp::Msk<N>` objects, respectively. The `T` and `N` template parameters correspond to the type and the number of elements inside the vector register and the mask register, respectively. In these registers, objects are typed, unlike in the MIPP low register basic type. This avoids to write the type when a MIPP function is called. The function type can then be directly selected from the parameter type. Listing 2.1 illustrates the template-based encapsulation, which enables MIPP to override common arithmetic and comparison operators.

MIPP med. also simplifies register loading and initialization operations. The constructor of the `mipp::Reg` object calls the `mipp::load` function automatically. Thus, a load in MIPP low:

```
mipp::reg a = mipp::load<float>(aligned_ptr);
```

can be simplified into:

```
mipp::Reg<float> a = aligned_ptr;
```

with MIPP med. level. An initializer list can be used with a MIPP med. vector register:

```
mipp::Reg<float> a = {1.f, 2.f, 3.f, 4.f};
```

Likewise, a scalar assigned to a vector sets all elements to this value.

2.1.3 Software Implementation Details

MIPP targets SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2, FMA3, KNCI, AVX-512F and AVX-512BW instruction sets on Intel® x86 and related architectures, as well as NEON, NEONv2, NEON64 and NEON64v2 on ARM®. It can easily be extended to other instruction sets. MIPP selects the most recent instruction set available at compile time. For instance, a code compiled with the `-march=avx` flag of the GNU GCC compiler uses AVX instructions even if the architecture supports SSE as well. The vector register size is determined by the instruction set and the data type. A dedicated function returns the number of elements in a MIPP register:

```
constexpr int n = mipp::nElmtsPerRegister<T>();
```

A shortened version of this previous function is also defined as: `mipp::N<T>()`. Whenever vectorization takes place in loops, MIPP's philosophy is to change the stride of the loop from one to the size of registers. The stride can be statically determined with the `mipp::N<T>()` function. If the loop size is not a multiple of the register size, 1) a sequential tail loop can be implemented to compute the remaining elements, 2) the padding technique can be implemented to force the loop size to be a multiple of the vector registers. When the instruction set cannot be determined, MIPP med. falls back on sequential instructions. In this case, MIPP does not use any intrinsic anymore. However, the compiler vectorizer still remains effective. This mode can also be selected by the programmer with the `MIPP_NO_INTRINSICS` macro.

MIPP supports the following data types: `double`, `float`, `int64_t`, `int32_t`, `int16_t` and `int8_t`. It also supplies an aligned memory allocator, that can be used with types such as the `std::vector<T,A>` vector container from the `C++` standard library (where `T` is the vector element type and `A` the allocator). The alignment requirements are not guaranteed by the default `C++` memory allocator. The MIPP memory allocator can be used as follows:

```
std::vector<T,mipp::allocator> aligned_data;
```

and shortened like this: `mipp::vector<T> aligned_data`.

MIPP comes with a comprehensive unitary test suite to validate new instruction set ports and new feature implementations. It has successfully been tested with the following minimum compiler versions: `g++-4.8`, `clang++-3.6`, `icpc15` and `msvc14.0`.

MIPP implements a generic reduction operator based on a reduction tree, which would be tedious to write by the application programmer, due to the sequence of heterogeneous shuffle instructions it implies. The computational complexity of this algorithm is $O(\log_2(N))$, with N the number of elements in a register. It can operate on `mipp::reg`, `mipp::Reg<T>` and `std::vector<T>`. It can also work on dynamically allocated arrays. It provides the length of the array that is a multiple of the vector register size. Since the function passed to the reduction operator is resolved at the compile time, the code remains efficient. Any function with the following prototype can be used as the reduction function:

```
mipp::Reg<T> func(mipp::Reg<T>, mipp::Reg<T>);
```

E.g., the code below computes the smallest element in a register:

```
mipp::Reg<float> r = {4.f, 2.f, 1.f, 3.f};
float min = mipp::Reduction<mipp::min>::sapply(r);
```

The `min` scalar variable will be assigned `1.f` as the result. For convenience, a set of functions is predefined, based on this generic reduction feature: `hadd`, `hsub`, `hmul` and `hdiv`.

2.1.4 Related Works

Many SIMD programming solutions to take advantage of conventional instruction sets have been surveyed in [Poh+16]. The existing alternatives can be decomposed into three main models: 1) intrinsics or assembly code; 2) dedicated language; 3) dedicated library. The intrinsics or assembly approaches are non-portable. Low-level solutions target specific architectures. They offer maximum control to take advantage of instruction set specificities, and to fine tune register usage. However, it is quite difficult to develop and maintain a low-level code in the long run. Some languages have been designed to provide programmers with SIMD programming constructs. Many of them are based on general purpose languages extended with some kinds of annotation mechanism (e.g. pragmas) such as OpenMP [Ope13], Cilk Plus [Rob13] or ispc [PM12]. They offer higher expressiveness, better portability and generally more readable code, at the expense of less programmer control, and vectorization performance. More specialized languages, such as OpenCL [How15], enable the programmer to retain more control, as the counterpart of writing some more specific code. In our study, the focus is given to the library approach since the main objectives are to maximize performance, maximize portability and deal with existing C++ codes.

2.1.4.1 C++ SIMD Wrappers

This section and the next one propose to compare different SIMD wrappers of the state-of-the-art. This comparison has been made in 2018 when we wrote a paper dedicated to MIPP [7]. Be aware that the features of the other wrappers presented here could have evolved since that time.

Table 2.1 and Table 2.2 compare various SIMD wrappers. Table 2.1 is focusing on the general features while Table 2.2 is targeting the supported instruction sets and data types. They aim to

present an overview of some prominent solutions, although they are by no means exhaustive due to the richness of the SIMD wrapper landscape. Some of the wrappers presented, such as MIPP, Vc, Boost.SIMD, VCL and T-SIMD, have been designed in an academic research context.

Table 2.1 – Comparison of various SIMD wrappers: General Information and Features.

General Information				**Features**		
Name	**Ref.**	**Start Year**	**License**	**Math Func.**	**C++ Technique**	**Test Suite**
MIPP	[7]	2013	MIT	✓	Op. overload.	✓
VCL	[Fog17]	2012	GNU GPL	✓	Op. overload.	**N/A**
simdpp	[Kan17]	2013	Boost Software	✗	Expr. templ.	✓
T-SIMD	[Möl16]	2016	Open-source	✗	Op. overload.	**N/A**
Vc	[KL12]	2012	BSD-3-Clause	✓	Op. overload.	✓
xsimd	[Mab17]	2014	BSD-3-Clause	✓	Op. overload.	**N/A**
Boost.SIMD	[Est+12b]	2012	Boost Software	✓	Expr. templ.	✓
bSIMD	[Est+12a]	2017	Non-free	✓	Expr. templ.	✓

Table 2.2 – Comparison of various SIMD wrappers: Supported ISA and Data Type.

Name	**Instruction Set**					**Data Type**					
	SSE 128-bit	**AVX** 256-bit	**AVX512** 512-bit	**NEON** 128-bit	**AltiV.** 128-bit	**Float** 64	32	**Integer** 64	32	16	8
MIPP	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
VCL	✓	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓
simdpp	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
T-SIMD	✓	✓	✗	✓	✗	✗	✓	✗	✓	✓	✓
Vc	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✗
xsimd	✓	✓	✗	✗	✗	✓	✓	✓	✓	✗	✗
Boost.SIMD	✓	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓
bSIMD	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Some others, simdpp and xsimd, appear to be standalone development efforts by individual programmers or maintainers. Proprietary, closed-source solutions also exist on the market, such as bSIMD, which is an extended version of Boost.SIMD, or the commercial version of VCL. The *Instruction Set* column is broken up into five families among the most widely available on the market: NEON, SSE, AVX, AVX-512 and AltiVec. For the sake of conciseness, we choose not to list all the instruction sets "sub-variants" (such as SSE2, SSE3, etc). simdpp et bSIMD propose the most comprehensive instruction set compatibility. At the other end of the range, xsimd and Boost.SIMD only support Intel® SIMD instruction sets. The *Data Type* column of the table summarizes the supported vector element types and precisions. In their public version, and at the time of writing, Vc does not support 8-bit integers, xsimd does not support 8-bit and 16-bit integers and T-SIMD does not support 64-bit data types, to the best of our knowledge. The *Features* column highlights some additional characteristics. The *Math Func.* column indicates which wrapper supports additional mathematical sub-routines, not necessarily available as native CPU instructions (exponential, logarithm, trigonometric functions for instance). These sub-routines are required by algorithms such as the Box-Muller Transform (see Section 2.3.1). The *C++ Technique* column indicates whether the wrapper is designed as an expression template framework, or whether it relies on operator overloading techniques. The expression template feature is a

powerful technique to automatically drive the rewriting of whole arithmetic expressions into SIMD hardware instructions or instruction sequences. For instance if the user writes `d = a * b + c`, the wrapper can automatically match a *fused multiply and add* instruction (FMA). Boost.SIMD and bSIMD extensively take advantage of this technique [Est+12b, Est+12a]. The drawbacks are that the source code complexity of the wrapper dramatically increases. Boost.SIMD and bSIMD have a dependency on the Boost framework to build, and currently available C++ compilers produce huge amounts of arcane error messages at the slightest mistake in the end user program. For these reasons, we decided not to base MIPP on the expression template technique. As mentioned in Section 2.1.3, maintaining SIMD wrappers, and porting them to new instruction sets is error prone by nature, due to the large number of routines, cryptic intrinsics names, and specific instruction set details. A comprehensive testing suite is therefore critical to validate new development, optimizations and ports on new instruction sets. This is why MIPP, as well as Vc, Boost.SIMD, simdpp and bSIMD come with their own test suites. We have not found similar test suites in the software distributions of VCL, xsimd and T-SIMD. However, test suites might be in use internally, within the development teams of these wrappers.

2.1.4.2 Experimentation Platforms

Table 2.3 – Specifications of the target processors.

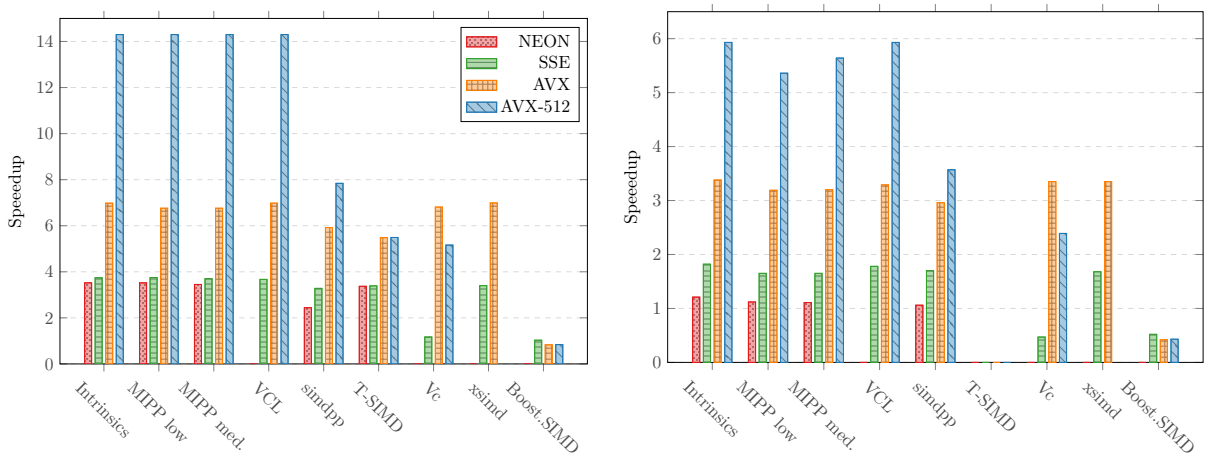
Name	Exynos5422	RK3399	Core™ i5-5300U	Xeon Phi™ 7230
Year	2014	2016	2015	2016
Vendor	Samsung®	Rockchip®	Intel®	Intel®
Arch.	ARMv7 Cortex-A15	ARMv8 Cortex-A72	Broadwell	Knights Landing
Cores/Freq.	4/2.0 GHz	2/1.6 GHz	2/2.3 GHz	64/1.3 GHz
LLC	2 MB L2	1 MB L2	3 MB L3	32MB L2
TDP	~4 W	~2 W	15 W	215 W

Four different architectures are considered for performance results as summarized in Table 2.3. The Cortex-A15 is used to evaluate the NEON instruction set in 32-bit. The Cortex-A72 is used to evaluate the 64-bit NEON instructions for Figure 2.1. The Core™ i5 is used for both SSE and AVX benchmarks. The Xeon Phi™ is used for AVX-512 instructions. Source codes are compiled with the GNU C++ 5 compiler using the common flags: `-O3 -funroll-loops`. The additional architecture specific flags are: 1) `-march=armv7-a -mfpu=neon-vfpv4` on Cortex-A15, 2) `-march=armv8-a` on Cortex-A72, 3) `-msse4.2` for SSE or `-mavx2 -mfma` for AVX on Core™ i5, 4) `-mavx512f -mfma` on Xeon Phi™. All experiments have been performed in single-threaded. All studied problem sizes fit into the last level cache (LLC) of CPUs. The references for the speedup computations are always sequential versions of the SIMD codes. Those reference versions can be auto-vectorized by the compiler, thus a reference version is compiled for each SIMD ISA.

2.1.4.3 Qualitative and Quantitative Comparisons

We now compare MIPP with the open-source wrappers presented above, both qualitatively for our error correction code purpose, and quantitatively on a well known benchmark. The computation of the Mandelbrot set is selected as the benchmark. It prevents as much as possible the risk of unfairness of the port on each wrapper. This problem is compute-bound. The chosen implementation relies on a floating-point representation (available online [2]). Figure 2.1

2. Mandelbrot set source code: https://gitlab.inria.fr/acassagn/mandelbrot



(a) Float 32-bit

(b) Float 64-bit

Figure 2.1 – Speedups over the Mandelbrot naive auto-vectorized implementation.

presents the speedups obtained on various instruction sets on single-precision floating-point format (see Figure 2.1a) and on double-precision floating-point format (see Figure 2.1b). SSE stands for SSE4.2, NEON stands for NEONv2 (includes the FMA instructions), AVX stands for AVX2+FMA3 and AVX-512 stands for AVX-512F (with FMA instructions). The FMA benefit ranges from 17% (AVX2) to 26% (AVX-512). An SIMD with intrinsics version has been hand-coded for each specific instruction set. The intrinsics version is considered the "*golden*" model.

Boost.SIMD only supports the SSE instruction set, even when the code is compiled with one of the AVX or AVX-512 flags. It is insufficient for our channel coding processing purpose. The Boost.SIMD wrapper performance results are disappointing because its contents has been migrated into the new proprietary bSIMD wrapper. The sequential Mandelbrot kernel does an early exit in the innermost loop, as soon as the divergence of the sequence is detected for the input coordinates. We were unable to SIMDize this early termination with Boost.SIMD, because the `boost::simd::any` function was not available in the GitHub repository at the time of writing.

xsimd achieves performance close to the intrinsic version in SSE and AVX. However, it currently lacks NEON and AVX-512 supports. Moreover, it does not support small 8-bit and 16-bit integers that are necessary for Successive Cancellation decoders (see Section 2.5).

Vc is one of the earliest developed SIMD C++ wrapper. We used Branch 1.3 for the performance measurements, the latest stable branch at this time. Vc includes a lot of of features compared to the other wrappers. However, it lacks support for NEON and AVX-512 (which are currently under progress). Performance results are on par with the best contenders for AVX. However, a slowdown is observed for SSE. For AVX-512, since the support is not yet available in the stable version, we used the capability of Vc to generate AVX2 code in order to produce the sample points for AVX-512 series. The results are likely to improve once the full AVX-512 support is released in a subsequent stable version.

T-SIMD is a wrapper primarily designed for image processing purpose. It performs well in 32-bit NEON, SSE and AVX. But, it lacks AVX-512. Support of the 64-bit types is not planned since it is not useful for image computations.

simdpp supports an impressive number of instruction sets. This may explain why it does not support mathematical functions so far. It matches the performance of the other wrappers for NEON and SSE, but falls behind for AVX, and even more for AVX-512.

VCL is a high performance wrapper and one of the most feature rich for x86 SIMD. It gives a lot of control to the developer and it is well documented. The obtained performance are on the same level as hand-written intrinsics. However, it is not yet available on NEON.

MIPP corresponds to a programming model close to the intrinsics, with some adaptations to architectures. Still, a high performance code requires that the developer knows how to decompose efficiently some computations with the SIMD instructions. Between AVX-512 and SSE or NEON for instance, several implementations of the same code are possible. MIPP offers to the programmer the control on the intrinsics selected and ensures portability. We have tested both the lower-level programming interface and the medium-level programming interface of our MIPP wrapper, mainly to detect potential overheads when using the medium level interface instead of the lower one. The obtained results do not show any performance penalties when using the MIPP medium level interface. Moreover, the obtained speedups are close to the intrinsics version.

2.2 Vectorization Strategies

Vectorization is a key feature to develop high performance implementations of signal processing algorithms. One of the main constraint of these algorithms is their low latency requirement. A typical signal processing latency requirement ranges between one microsecond to a few nanoseconds for a frame. Therefore, the usual multi-threading parallelism is not well-suited to speedup a single signal processing algorithm. Indeed, the threads synchronizations overhead is too high compared to the expected latency of these algorithms. Moreover, generally the signal processing algorithms are implemented on hardware targets (ASIC or FPGA). Consequently, these algorithms have been refined to take advantage of the fixed-point arithmetic. It enables software implementations working on 16-bit and 8-bit integers. Combining the SIMDization and the fixed-point arithmetic leads to a high level of SIMD parallelism. For instance, if we consider the AVX-512 ISA, there are 512 bits in the registers. If computations are made on 8-bit integers, the available parallelism is $\mathrm{p}_{\mathrm{SIMD}}^{\text{8-bit}} = 512/8 = 64$. This is more than the number of cores that is available in most of the current CPUs. The multi-threading technique will be used later in the document (see Section 3.4.1, Section 4.6.3 and Chapter 5), at a higher level, to parallelize chains of signal processing algorithms. The next subsections detail the vectorization strategies that we have identified to effectively implement signal processing algorithms on CPUs.

2.2.1 Intra-frame SIMD Strategy

The *intra-frame* SIMD strategy consists in using vectorization to process a single frame. With this strategy, the available level of parallelism depends on the characteristics of the signal processing algorithm. It is up to the developer to clearly identify the inherent parallelism of the algorithm and to map it on SIMD instructions. One of the main advantages of this method is that the latency of the processing can be divided, at best, by the parallelism factor $\mathrm{p}_{\mathrm{SIMD}}$. Thus, the intra-frame SIMD strategy is a key of very low latency signal processing implementations on CPUs. Note that the throughput can also be increased by a factor of $\mathrm{p}_{\mathrm{SIMD}}$. However, a limitation of the *intra-frame* strategy is when the algorithm inherent parallelism is lower than $\mathrm{p}_{\mathrm{SIMD}}$. In this specific case, SIMD instructions cannot be used on full vector registers and some

efficiency is loss. Even if the inherent parallelism is higher than p_{SIMD} the targeted algorithm can require data movements inside the SIMD registers and cause some shuffle and permutation extra-instructions have to be added. These extra instructions are generally a limiting factor in the efficiency of the intra-frame SIMD strategy.

2.2.2 Inter-frame SIMD Strategy

Unlike the intra-frame SIMD strategy, the *inter-frame* SIMD strategy processes several frames in parallel. The idea is to fill the SIMD registers with data coming from multiple frames. If there is a p_{SIMD} parallelism, then $\mathsf{F} = \mathsf{p}_{\text{SIMD}}$ frames are used to fill the SIMD registers. The main advantage of this strategy is that the effective level of parallelism does not depend on the algorithm. In other words, it is always possible to use 100% of the SIMD registers. Thus, the inter-frame SIMD strategy is a key of very high throughput signal processing implementations on CPUs. However, this technique does not reduce the overall latency of the processing as it computes $\mathsf{F} = \mathsf{p}_{\text{SIMD}}$ in parallel, the throughput can be increased by a factor p_{SIMD} but the latency of a single frame is mainly unchanged.

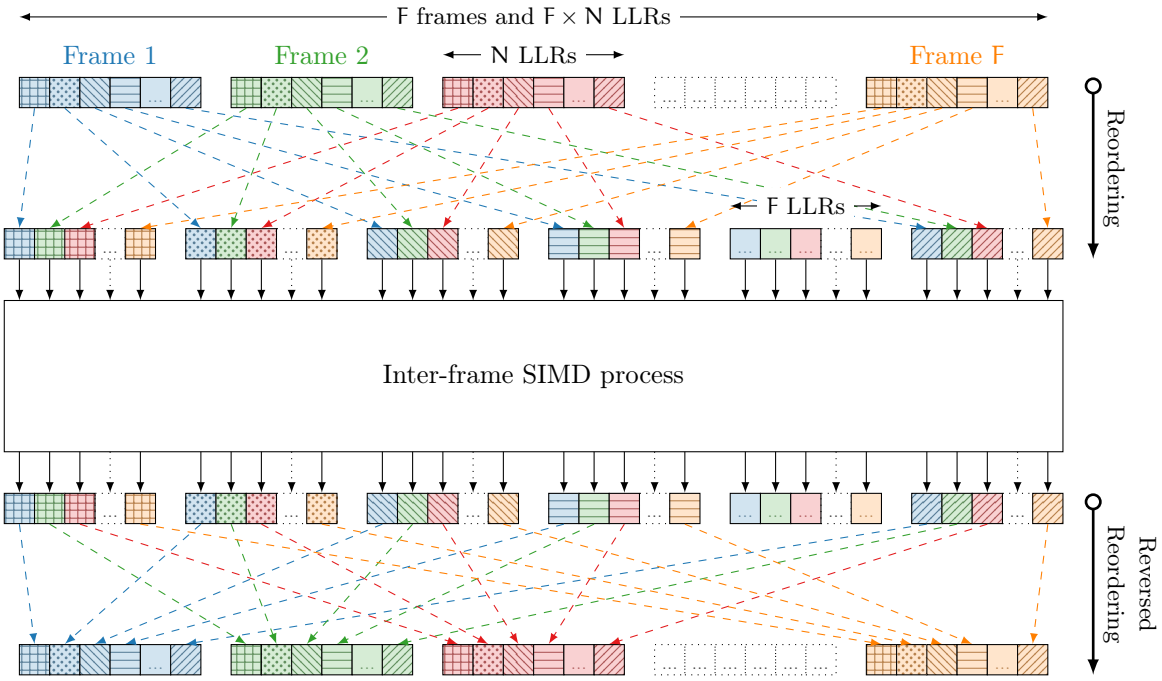


Figure 2.2 – Frame reordering operation before and after an inter-frame SIMD process.

Moreover, one may note that the N bits of a frame are naturally transmitted frame by frame. It is then necessarily to perform a reordering operation before to start the inter-frame SIMD computations. This reordering process is illustrated in Figure 2.2 where F frames are represented by different colors. The elements inside a single frame can be identified by the different patterns. All the first elements of each frame are regrouped together in memory, then all the second elements of each frame are regrouped together, and so on. The dual reverse reordering operation is performed at the end of the inter-frame SIMD computations. The reordering operations are the main key to reach high throughputs in the inter-frame SIMD strategy. They enable efficient load and store instructions. These operations could be replaced by gather and scatter instructions each time a load and store instructions are required. But this solution delivers poorer performance on current CPU architectures. The selected version with reordering operations comes with a much better data locality at the cost of extra operations before and after the SIMD computations.

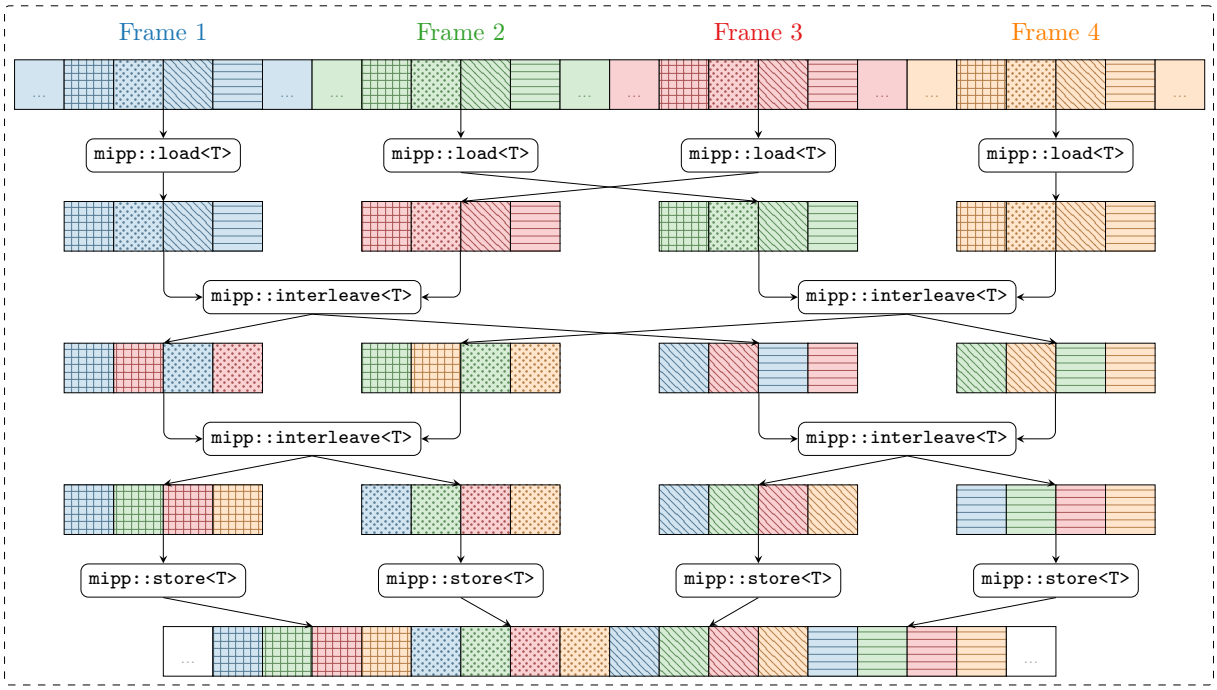


Figure 2.3 – MIPP implementation of the SIMD frame reordering process for $\mathrm{p_{SIMD}} = 4$.

It is then mandatory to minimize the overhead introduced by the reordering operations. To this purpose we developed a SIMD reordering function that extensively uses the `mipp::interleave` function to reorder the elements. This function is completely vectorized and generic on the data types and the frame sizes. The `mipp::interleave` function takes two registers $\mathbf{r^a} = [r^a_0, r^a_1, ..., r^a_{\mathrm{p_{SIMD}}}]$ and $\mathbf{r^b} = [r^b_0, r^b_1, ..., r^b_{\mathrm{p_{SIMD}}-1}]$ as input parameters and returns two output registers $\mathbf{r^c} = [r^a_0, r^b_0, ..., r^a_{\mathrm{p_{SIMD}}/2-1}, r^b_{\mathrm{p_{SIMD}}/2-1}]$ and $\mathbf{r^d} = [r^a_{\mathrm{p_{SIMD}}/2}, r^b_{\mathrm{p_{SIMD}}/2}, ..., r^a_{\mathrm{p_{SIMD}}-1}, r^b_{\mathrm{p_{SIMD}}-1}]$. The `mipp::interleave` function is applied $\mathsf{N} \times \log_2(\mathsf{F})$ times. Figure 2.3 shows the MIPP implementation of the reordering process for $\mathsf{F} = \mathrm{p_{SIMD}} = 4$. To increase the speed of the reordering process, the template meta-programming technique has been used to unroll the `mipp::interleave` calls. The generic reversed reordering process has also be implemented with MIPP and the principle is similar to the reordering process.

2.2.3 Intra-/inter-frame SIMD Strategy

The *intra-/inter-frame* SIMD strategy is the combination of the two previous strategies. For instance, if we have a hardware parallelism $\mathrm{p_{SIMD}} = 16$ and if the algorithm has only an inherent parallelism of 8, then it is possible to use the intra-frame SIMD strategy to absorb the parallelism of the algorithm and to use the inter-frame SIMD strategy on two frames. The intra-/inter-frame SIMD strategy is also a good candidate to make trade-offs between low latency and high throughput.

2.3 Efficient Functional Simulations

This section focuses on computational blocks specific to the functional simulation of digital communication systems. In this type of simulations, the channel model can take a non negligible amount of time. This is why we first propose a vectorized version of the AWGN channel. Additionally when fixed-point decoders are benched, the output LLRs of the demodulator have to

be converted from a floating-point representation to a fixed-point representation. This process can also take an important amount of time and has been optimized. The proposed implementations are briefly evaluated with the protocol defined in Section 2.1.4.2.

2.3.1 Box-Muller Transform

Monte Carlo simulations of digital communication systems provide an empirical way to evaluate error correction performance of the digital system. In this kind of simulations, the transmission channel is modeled as a white Gaussian noise added to the modulated data. This noise generation can be split in two parts: 1) the uniformly-distributed random variable generation, 2) the transformation to a Gaussian random variable. An uniform noise can be generated by a pseudo random number generator (PRNG) like the Mersenne Twister 19937 [MN98] (MT19937). Then, the Box-Muller method [BM58] transforms uniformly distributed random numbers into normally distributed random numbers.

Suppose $\mathcal{U}_1$ and $\mathcal{U}_2$ are independent random variables uniformly distributed in $]0, 1]$:

$$z_1 = \sqrt{-2\log \mathcal{U}_1}.\cos(2\pi.\mathcal{U}_2),\ z_2 = \sqrt{-2\log \mathcal{U}_1}.\sin(2\pi.\mathcal{U}_2).$$

Then, z_1 and z_2 are independent and normally distributed samples.

```
#include <mipp.h>

void box_muller_transform(const std::vector<float> &uni_rand,
                          std::vector<float> &norm_rand) {
    const size_t N = uni_rand.size();
    const float two_pi = 2.f * 3.141592f;
    for (size_t n = 0; n < N; n += mipp::N<float>() * 2)
    {
        const auto u1 = mipp::Reg<float>(&uni_rand[                  n]);
        const auto u2 = mipp::Reg<float>(&uni_rand[mipp::N<float>() +n]);
        const auto radius = mipp::sqrt(mipp::log(u1) * -2.f);
        const auto theta  = u2 * two_pi;
        mipp::Reg<float> sintheta, costheta;
        mipp::sincos(theta, sintheta, costheta);
        auto z1 = radius * costheta;
        auto z2 = radius * sintheta;
        z1.store(&norm_rand[                  n]);
        z2.store(&norm_rand[mipp::N<float>() +n]);
    }
}
```

Listing 2.2 – Box-Muller Transform SIMD kernel with MIPP.

Listing 2.2 presents a new MIPP implementation of the Box-Muller transform. `uniRand` is a vector of independent and uniformly distributed random numbers. For instance, it can be generated with the MT19937 PRNG. `norRand` is a vector of independent and normally distributed random numbers. The code stresses SIMD units with multiplications, `mipp::sqrt` and `mipp::sincos` calls. The $\mathcal{U}_1$ and $\mathcal{U}_2$ are independent variables. So the algorithm inherent parallelism directly depends on the frame size N. Generally $\mathsf{N} > \mathsf{p}_{\text{SIMD}}$, this is why the proposed implementation is based on the intra-frame SIMD strategy.

Table 2.4 – AWGN channel throughputs and speedups of the MIPP implementation.

	NEON	SSE	AVX	AVX-512
SIMD size	4	4	8	16
Throughput (Mb/s)	40.9	107.4	178.3	95.1
Speedup	×3.1	×2.3	×4.2	×14.4

Table 2.4 presents the measured speedups with the same MIPP code compiled for NEON, SSE, AVX and AVX-512, compared to the sequential code (can be auto-vectorized). It also gives the actual throughput for each instruction set. The proposed kernel is compute intensive and the speedups are mainly driven by the MIPP implementations of the `mipp::sincos` trigonometric function. The conversion of floating-point format from single precision to double precision only requires to replace the `float` keyword by `double`. The ability to switch seamlessly from one data type to another is clearly a strength of the MIPP library.

2.3.2 Quantizer

During the implementation of ECC decoders, a common step is to convert the floating-point representation into a fixed-point representation. This is necessary after the reception of the noisy channel information representing *Logarithmic Likelihood Ratios* (LLRs) and encoded as real values. The reduction of the LLRs precision (from 32 bits floating-point to 16 or 8 bits fixed-point) does not significantly affect error correction performance. But, it provides more SIMD parallelism. The quantizer computes:

$$l^n_{s,v} = \min(\max(2^v.l^n \pm 0.5, -2^{s-1}+1), 2^{s-1}-1), \tag{2.1}$$

with l^n the current floating-point value, s the number of bits of the quantized number, including v bits for the fractional part.

```
1  void quantize_seq(const std::vector<float > &l_float,
2                    std::vector<int8_t> &l_fixed,
3                 const uint32_t s, const uint32_t v) {
4      const size_t N = l_float.size();
5      const float factor = 1 << v;
6      assert(s >= 2);
7      const float q_max = (1 << (s-2)) + (1 << (s-2)) -1;
8      const float q_min = -q_max;
9      for (size_t n = 0; n < N; n++)
10     {
11         // q = 2^v * y +- 0.5
12         const float q = std::round(factor * l_float[n]);
13         // saturation
14         l_fixed[n] = (int8_t)std::min(std::max(q, q_min), q_max);
15     }
16 }
```

Listing 2.3 – Sequential implementation of the quantizer.

The associate sequential code is presented in Listing 2.3. The code converts `float` (32-bit floating-point number) to `int8_t` (8-bit signed integer). Although the scalar code is fairly simple, the compiler fails to auto-vectorize the `for-loop` [7]. MIPP enables to convert floating-point data types to integers with the `mipp::cvt` function. It also compresses larger data types into shorter

```cpp
#include <mipp.h>

void quantize_simd(const std::vector<float > &l_float,
                         std::vector<int8_t> &l_fixed,
                   const uint32_t s, const uint32_t v) {
    const size_t N = l_float.size();
    const float factor = mipp::Reg<float>(1 << v);
    assert(s >= 2);
    const float q_max = (1 << (s-2)) + (1 << (s-2)) -1;
    const float q_min = -q_max;
    for (size_t n = 0; n < N; n += 4 * mipp::N<float>())
    {
        // implicit loads and q = 2^v * y +- 0.5
        mipp::Reg<float> q32_0 = mipp::round(factor * &l_float[n+0*mipp::N<float>()]);
        mipp::Reg<float> q32_1 = mipp::round(factor * &l_float[n+1*mipp::N<float>()]);
        mipp::Reg<float> q32_2 = mipp::round(factor * &l_float[n+2*mipp::N<float>()]);
        mipp::Reg<float> q32_3 = mipp::round(factor * &l_float[n+3*mipp::N<float>()]);
        // convert float to int32_t
        mipp::Reg<int32_t> q32i_0 = mipp::cvt<int32_t>(q32_0);
        mipp::Reg<int32_t> q32i_1 = mipp::cvt<int32_t>(q32_1);
        mipp::Reg<int32_t> q32i_2 = mipp::cvt<int32_t>(q32_2);
        mipp::Reg<int32_t> q32i_3 = mipp::cvt<int32_t>(q32_3);
        // pack four int32_t in two int16_t
        mipp::Reg<int16_t> q16i_0 = mipp::pack<int32_t,int16_t>(q32i_0, q32i_1);
        mipp::Reg<int16_t> q16i_1 = mipp::pack<int32_t,int16_t>(q32i_2, q32i_3);
        // pack two int16_t in one int8_t
        mipp::Reg<int8_t> q8i = mipp::pack<int16_t,int8_t>(q16i_0, q16i_1);
        // saturation
        mipp::Reg<int8_t> q8is = mipp::sat(q8i, q_min, q_max);
        q8is.store(&l_fixed[k]);
    }
}
```

Listing 2.4 – SIMD implementation of the quantizer with MIPP.

Table 2.5 – Quantizer throughputs and speedups of the MIPP implementation.

	NEON	**SSE**	**AVX**
SIMD size	4-16	4-16	8-32
Throughput (Mb/s)	300.6	3541.4	5628.3
Speedup	×4.6	×15.6	×25.8

ones with the `mipp::pack` function. The MIPP code is presented in Listing 2.4. It performs explicit data types packaging, while in the sequential code, this operation is done implicitly by the `(int8_t)` cast. Table 2.5 summarizes the obtained speedups with MIPP. For this specific case study the speedups are significant for SSE and AVX. They are less important with the NEON instruction set but still non-negligible. We do not provide results for AVX-512, since an AVX-512BW compatible CPU would be required and the Xeon Phi™ 7230 is not.

2.4 LDPC Decoders

The LDPC decoders have been extensively studied in the literature. In this section we focus on the Belief Propagation (BP) algorithm and its variants. On CPUs, the inter-frame SIMD strategy has been explored and leads to very high throughputs on many different LDPC codes [LJ16]. On the other hand, the intra-frame strategy has also be studied [LJ19, Xu+19], the achieved throughputs are lower compared to the inter-frame SIMD strategy but the latencies are also significantly lower. To the best of our knowledge, the implementations proposed in the literature always target a specific combination of scheduling and update rules. We found this is a limitation to explore new trade-offs. Our contribution is to propose a generic implementation with a clear separation of concerns between the scheduling and the updates rules.

2.4.1 Generic Belief Propagation Implementation

The intra- and inter-frame SIMD strategies have been considered and the inter-frame strategy has been adopted. The reason of this choice is that the intra-frame implementations of the BP are directly depending on the $\mathcal{H}$ parity matrix. In many current standards using LDPC codes, the $\mathcal{H}$ parity matrices are *Quasi-Cyclic* (QC). QC $\mathcal{H}$ parity matrices are matrices that are composed of an array of $\mathsf{Z} \times \mathsf{Z}$ circulant identity sub-matrices. Z is the order that defines the parallelism level. So, the intra-frame SIMD parallelism directly depends on the $\mathcal{H}$ parity matrix and whether the parity matrix is QC or not. Even if the selected $\mathcal{H}$ matrix is QC, the Z inherent parallelism can be lower than $\mathfrak{p}_{\text{SIMD}}$ or higher but not a multiple of $\mathfrak{p}_{\text{SIMD}}$. At the end we conclude that the intra-frame SIMD strategy is not a good candidate for a generic software implementation.

Our main motivation in the proposed implementation is to be able to write the scheduling description (BP-F, BP-HL, BP-VL, etc) once and to combine this scheduling with any update rules (SPA, MS, OMS, NMS, etc.) written once too. The implementation of the BP horizontal layered (BP-HL) scheduling is given in Listing 2.5. `var_nodes` is the vector of LLRs in the variable nodes ($\boldsymbol{\nu}$). `contribs` is the vector of the a priori information ($\mathfrak{a}$) and `messages` is the vector of the extrinsic LLRs ($\boldsymbol{\epsilon}$). In the horizontal layered scheduling, the extrinsic LLRs can be updated during a single iteration. This is the key of a faster convergence compared to the flooding scheduling. In the presented `decode_single_ite` method, one can note that inner methods are called on the `up_rules` member. The `up_rules` member is an object of `Update_rules` class. This class is generic and is given as a template parameter. This way, the traditional inheriting scheme is bypassed. This choice has been made to improve the efficiency of the decoder. Using inheritance would have prevented the compiler to inline method calls on the `up_rules` member. The `Update_rules` class can be all the update rules such as the SPA, the MS, the OMS, the NMS, etc. An `Update_rules` class has to implement the interface composed by the `begin_chk_node_in`, `compute_chk_node_in`, `end_chk_node_in`, `begin_chk_node_out`, `compute_chk_node_out` and `end_chk_node_out` methods.

```cpp
template <typename B, typename R, class Update_rules>
void Decoder_LDPC_BP_horizontal_layered<B,R,Update_rules>
::decode_single_ite(std::vector<R> &var_nodes, std::vector<R> &messages) {
    size_t kr = 0, kw = 0; // read and write indexes
    const auto n_chk_nodes = (int)this->H.get_n_cols();
    for (auto c = 0; c < n_chk_nodes; c++)
    {
        const auto chk_degree = (int)this->H[c].size();
        this->up_rules.begin_chk_node_in(c, chk_degree);
        for (auto v = 0; v < chk_degree; v++)
        {
            this->contribs[v] = var_nodes[this->H[c][v]] - messages[kr++];
            this->up_rules.compute_chk_node_in(v, this->contribs[v]);
        }
        this->up_rules.end_chk_node_in();

        this->up_rules.begin_chk_node_out(c, chk_degree);
        for (auto v = 0; v < chk_degree; v++)
        {
            messages[kw] = this->up_rules.compute_chk_node_out(v, this->contribs[v]);
            var_nodes[this->H[c][v]] = this->contribs[v] + messages[kw++];
        }
        this->up_rules.end_chk_node_out();
    }
}
```

Listing 2.5 – LDPC BP-HL scheduling implementation.

```cpp
template <typename R> void Update_rules_MS<R>
::begin_chk_node_in(const int chk_id, const int chk_degree) {
    this->sign = mipp::Msk<mipp::N<R>()>(false);
    this->min1 = mipp::Reg<R>(std::numeric_limits<R>::max());
    this->min2 = mipp::Reg<R>(std::numeric_limits<R>::max());
}
template <typename R> void Update_rules_MS<R>
::compute_chk_node_in(const int var_id, const mipp::Reg<R> var_val) {
    const auto var_abs  = mipp::abs(var_val);
    this->sign ^= mipp::sign(var_val);
    this->min2  = mipp::min(this->min2, mipp::max(var_abs, this->min1));
    this->min1  = mipp::min(this->min1,           var_abs             );
}
template <typename R> void Update_rules_MS<R>
::end_chk_node_in() {
    this->cst1 = mipp::max(mipp::Reg<R>(0), this->min2);
    this->cst2 = mipp::max(mipp::Reg<R>(0), this->min1);
}
template <typename R> mipp::Reg<R> Update_rules_MS<R>
::compute_chk_node_out(const int var_id, const mipp::Reg<R> var_val) {
    const auto var_abs = mipp::abs(var_val);
          auto res_abs = mipp::blend(this->cst1, this->cst2, var_abs == this->min1);
    return mipp::copysign(res_abs, this->sign ^ mipp::sign(var_val));
}
```

Listing 2.6 – LDPC MS update rules implementation.

The implementation of the MS update rules is given in Listing 2.6. For the sake of conciseness, the `begin_chk_node_out` and `end_chk_node_out` methods are not shown. These methods are empty in the MS update rules. Each operation in the methods of the `Update_rules_MS` class is vectorized thanks to MIPP. It results in a readable source code close to a traditional `C++` code where the `std` namespace has been replaced by the `mipp` namespace.

Both the scheduling and update rule implementations have a generic real `R` type. This way the same BP decoder can automatically adapt to floating-point or fixed-point representations of the LLRs values. The generic `B` type is for the decoder output bits representation.

Before to start the decoding process, the reordering step explained in Section 2.2.2 is performed. After the last decoding iteration, the output bits are decided from the updated variable nodes ($\boldsymbol{\nu}$) and the input LLRs ($\mathbf{l}$). Finally the reversed reordering is performed to recover the natural order of the frames.

We implemented the flooding, the horizontal layered and the vertical layered scheduling as well as the SPA, log-SPA, MS, OMS, NMS and AMS update rules. The main advantage of the proposed methodology is to ease the creation of new scheduling strategies and update rules without sacrificing too much the throughput performance (see Section 4.1). It is then possible to quickly evaluate the error-rate performance of a new algorithm and to combine it with all the previously implemented strategies.

2.4.2 Specialized Belief Propagation Implementation

Unfortunately, even if we observed that the function calls in the `decode_single_ite` method are effectively inlined by the compiler, we were not able to remove some useless memory loads and stores. These extra memory loads and stores are the consequence of the compiler limited knowledge on the application. As the horizontal layered scheduling and the MS, NMS and OMS update rules are often used, we decided to specialize a decoder for them. The specialized decoder consists in the merge of the scheduling class and the update rules classes. In this work, the merge operation has been done manually but it could be interesting to automate this process. The throughput and latency performances of this decoder are evaluated in Section 4.1.

2.5 Polar Decoders

The polar decoder algorithms presented in Section 1.3.3 has a number of characteristics of interest for its optimization:

— The tree traversal is sequential. `f`, `g` and `h` (see Equation 1.7) functions are applied element-wise to all elements of the LLR and bits in the nodes and their children. As there is no dependence between computations involving different elements of the same node, these node computations can be parallelized or vectorized (see the intra-frame SIMD strategy introduced in [Gia+14]);

— Frozen bits fully define their leaf values. Hence some parts of the traversal can be cut and their computations avoided, depending on the location of the frozen bits as introduced in Section 1.3.3.4;

— The decoder software implementation can be specialized for a particular configuration of frozen bits, as frozen bit locations do not change for many frames;

— Multiple frames can be decoded concurrently with vector code. Such inter-frame optimiza-

tions can increase the decoding throughput. However these optimizations are obtained at the expense of latency, which is also an important metric of the application (see [LLJ15]).

Beside optimizations coming from the computations in the tree, several representations of LLR may lead to different error correction performance. A LLR for instance can be represented by floats or integers (fixed point representation). Moreover, LLRs from different frames can be packed together. Finally, usual code optimizations, such as unrolling or inlining can also be explored. For instance, the recursive structure of the tree computation can be fully flattened, depending on the size of the code length.

2.5.1 Tree Pruning Strategy

2.5.1.1 Pattern Matching Algorithm

In order to perform the polar tree pruning, we implemented a pattern matching algorithm. For instance, knowing the initial location of the frozen bits, this algorithm can generate the pruned tree structure as illustrated in Figure 1.9. Each internal node has a tag indicating the type of processing required at that node (recursive children processing, f/g/h functions to be applied or not). This tag is initially set to *any*.

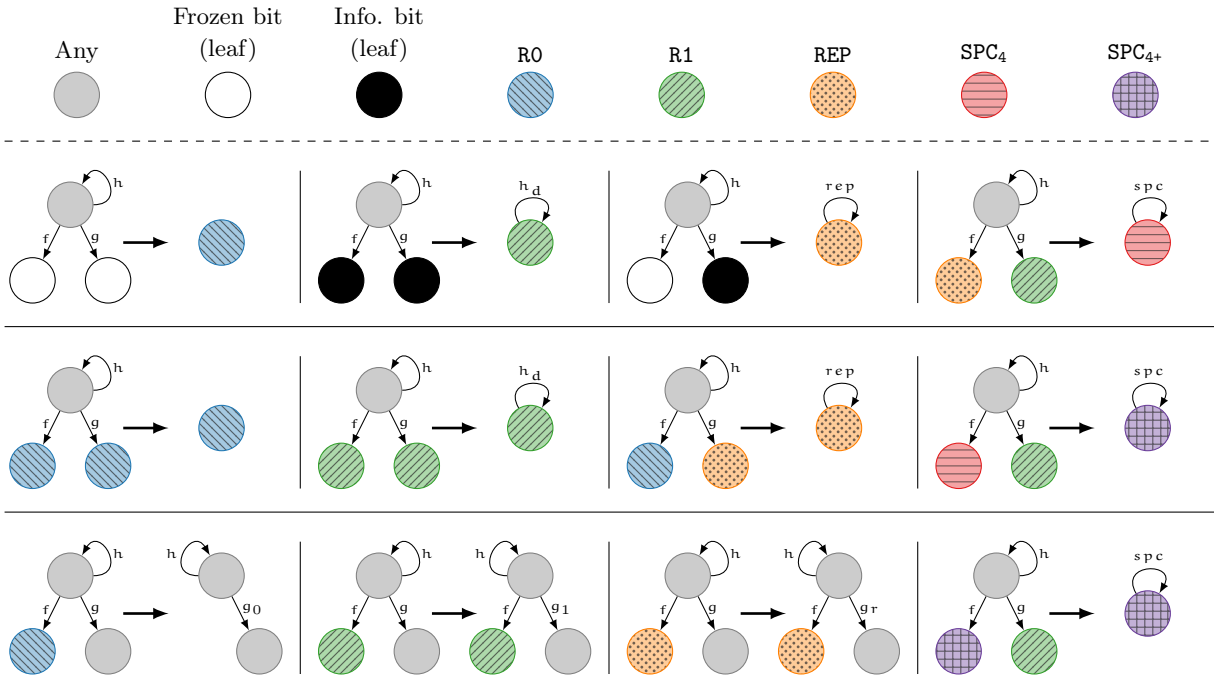


Figure 2.4 – Polar sub-tree rewriting rules for processing specialization.

For some sub-tree pattern configurations, the processing to be performed at the root of such sub-trees can be simplified, or even skipped completely, for instance, when a node only has two frozen bit leaf children. To exploit such properties, the decoder generator repeatedly applies the set of sub-tree rewriting rules listed in Figure 2.4 using a depth first traversal to alter the node tags, until no rewriting rule applies anymore.

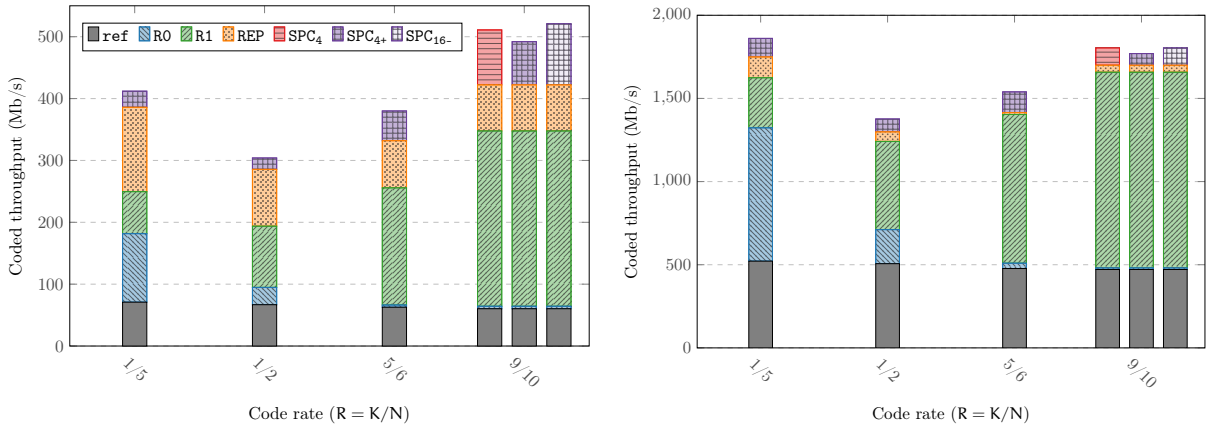
Each rewriting rule defines a subtree pattern *selector*, a new *tag* for the subtree root, and the f, g, and h *processing functions* to be applied, simplified or skipped for this node in the resulting decoder. A *null* f (resp. g) function cuts the left (resp. right) child of the node. From an implementation point of view, a rule is defined as a class, with a `match` function and a set of

functions f, g, and h. The current set of rewriting rules can thus easily be enriched with new rules to generate even more specialized versions.

Patterns on the first two rows result in cutting away both children. For instance, the first rule cuts the two frozen bit leaf children of the parent node, and tags it as R0 (blue node). Processing is completely skipped on this node since the values of the bits are unconditionally known. The REP rules match subtrees where only the rightmost leaf is black, the others being frozen bits. In this case, the whole subtree is cut and replaced by a simpler processing. Moreover a single, specialized rep function is applied on the node instead of the three functions f, g and h. The three first rules of the third line describe partial cuts and specialization. For instance, the third rule of the third column specializes the g functions in g_r, but does not prune the recursive children processing.

Rewriting rules are ordered by decreasing priority (left to right, then top row to bottom row in Figure 2.4). Thus, if more than one rule match an encountered subtree, the highest priority rule is applied. The priority order is chosen such as to favor strongest computation reducing rules over rules with minor impact, and to ensure confluence by selecting the most specific pattern first. Rule selectors can match on node tags and/or node levels (leaf, specific level, above or below some level). A given rule is applied at most once on a given node.

2.5.1.2 Impact of the Tree Pruning on the Decoding Performances



(a) 32-bit intra-frame SIMD (AVX)

(b) 8-bit inter-frame SIMD (SSE4.1)

Figure 2.5 – Throughput of the SSC decoder depending on the different optimizations for N = 2048, intra-frame vectorization on the left and inter-frame vectorization on the right, resp. (on the Intel® Xeon™ E31225 CPU).

The tree pruning step has a dramatical effect in general. For instance, on the SSC decoder, the reference code for a rate of 1/2 has 2047 nodes, whereas only 291 nodes remain in the pruned version. However, the individual effect of each rewriting rule is not trivial. The plots in Figure 2.5 show the specific impact of several rewriting rules (R0, R1, REP and SPC), with N = 2048 and multiple code rates, for intra-frame and inter-frame vectorization, respectively. The purpose of the plots is to show that no single rewriting rule dominates for each code rate. They also show that the respective impact of each rule may vary a lot from rate to rate, making the case for the flexible, extensible architecture proposed. Indeed, the rewriting rule set can also be enriched with rules for specific ranges of code rates. For instance, the rule *Single Parity Check* (SPC) has been applied with different level limits for 9/10 code rate, where it has a significant impact and may benefit from fine tuning.

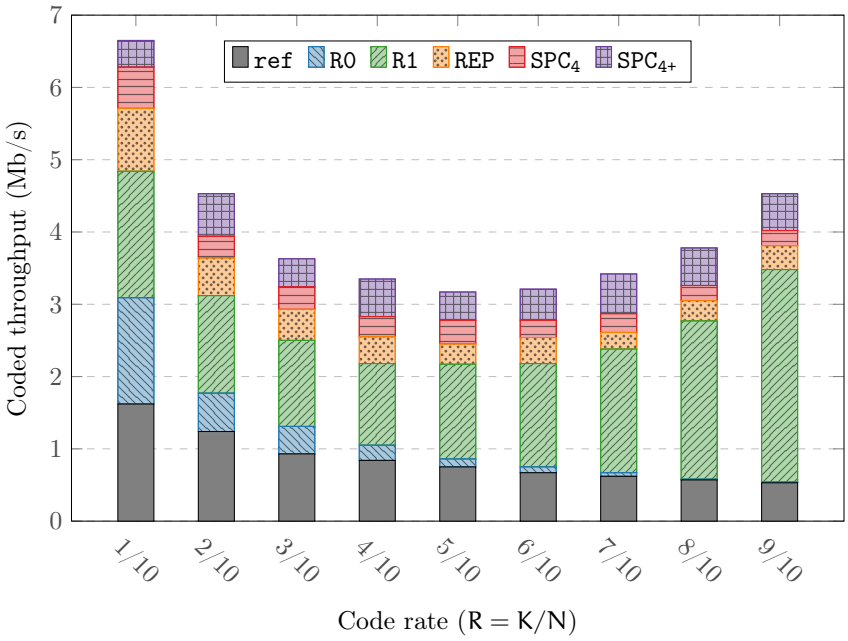


Figure 2.6 – Impact of the specialized nodes on the SSCL coded throughput. 32-bit intra-frame vectorization, N = 2048 and L = 32 (on the Intel® Core™ i5-6600K CPU).

Figure 2.6 shows the impact of the different tree pruning optimizations on the SSCL decoder throughput according to the code rate. The performance improvements are cumulative. Coded throughput, in which the redundant bits are taken into account, is shown instead of information throughput, for which only information bits are considered. It illustrates the computational effort without the influence of the fact that higher rate codes involve higher information throughputs.

The coded throughput of the original unpruned algorithm (`ref`) decreases as the code rate increases. Indeed, frozen bit leaf nodes are faster to process than information bit leaf nodes, in which a threshold detection is necessary. As there are more `R0` and `REP` nodes in low code rates, the tree pruning is more efficient in the case of low code rates. The same explanation can be given for `R1` nodes in high code rates. `R1` node pruning is more efficient than `R0` node pruning on average. Indeed, a higher amount of computations is saved in `R1` nodes than in `R0` nodes.

It has also been observed in [Sar+16] that when the `SPC` node size is not limited to 4, the decoding performance may be degraded. Consequently the size is limited to 4 in $\texttt{SPC}_4$. For $\texttt{SPC}_{4+}$ nodes, there is no size limit. The two node types are considered in Figure 2.6. Therefore, the depth at which dedicated nodes are activated in the proposed decoder can be adjusted, in order to offer a tradeoff between throughput and decoding performance.

According to our experiments, the aforementioned statement about performance degradation caused by $\texttt{SPC}_{4+}$ nodes is not always accurate depending on the code and decoder parameters. The impact of switching *on* or *off* $\texttt{SPC}_{4+}$ nodes on decoding performance and throughput at a FER of 10^{-5} is detailed in Figure 2.7. It shows that $\texttt{SPC}_{4+}$ nodes have only a small effect on the decoding performance. With L = 8, an SNR degradation lower than 0.1 dB is observed, except for one particular configuration. Throughput improvements from 8 to 23 percents are observed. If L = 32, the SNR losses are more substantial (up to 0.5 dB), whereas throughput improvements are approximately the same. Besides this observation, Figure 2.7 shows how the proposed decoder flexibility enables to easily optimize the decoder tree pruning, both for software implementations and for hardware implementations in which tree pruning can also be applied [LXY14].

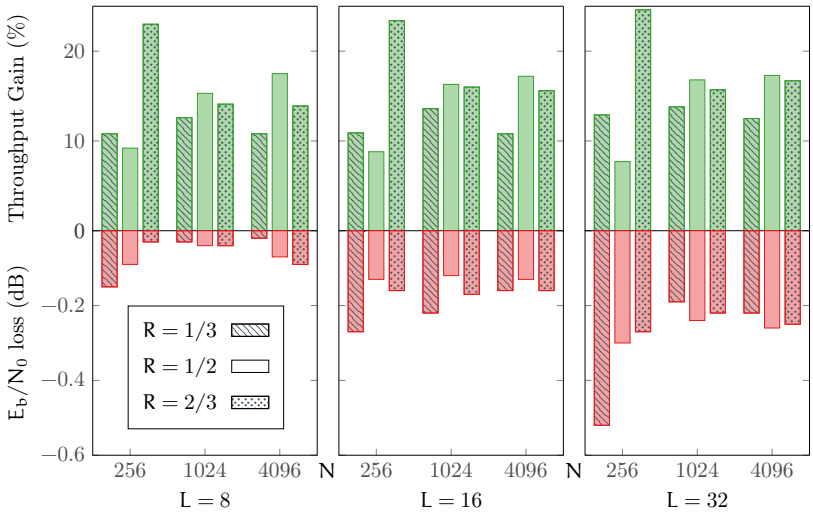


Figure 2.7 – Effects of the $\texttt{SPC}_{4+}$ nodes on the CA-SSCL decoder @ 10^{-5} FER. 32-bit intra-frame SIMD strategy (on the Intel® Core™ i5-6600K CPU).

2.5.2 Polar Application Programming Interface

The main challenge in implementing an architecture dependent Application Programming Interface (API) is to provide enough flexibility to enable varied types, data layout and optimization strategies such as intra-frame SIMDization (intra-SIMD) and inter-frame SIMDization (inter-SIMD), without breaking the high level skeleton abstraction. To meet this requirement, our API heavily relies on generic programming and compile time specialization by the means of `C++` templates, in a manner inspired by *expression template* techniques [Str13]. Template specializations provide node functions.

Reducing the decoding time with SIMD instructions is a classical technique in former software polar decoder implementations. The proposed polar decoders are based on specific building blocks included from the Polar API [4, 5]. These blocks are fast optimized implementations of the `f`, `g`, `h` (and their variants) polar intrinsic functions defined in Equation 1.7. Listing 2.7 details the SIMD implementation of these functions. These implementations are based on MIPP. Consequently, the description is clear, portable, multi-format (32-bit floating-point, 16-bit and 8-bit fixed-points) and as fast as an architecture specific code. The `mipp::Reg<B>` and `mipp::Reg<R>` types correspond to SIMD registers. `B` and `R` define the type of the elements that are contained in this register. `B` for *bit* could be `int`, `short` or `char`. `R` for *real* could be `float`, `short` or `char`. In Listing 2.7, each operation is made on several elements at a time. For instance, line 17, the addition between all the elements of the `neg_la` and `lb` registers is executed in a single CPU cycle. We also tried an auto-vectorized approach but even if all the routines were well vectorized (from the GCC 5.4 compiler report), the performance was, at least, 3 times slower than the MIPP handwritten versions. The template `N_ELMTS` parameter is not used in the proposed `API_polar_SIMD` implementation. The interest of this parameter will be explained in the next section.

A single SIMD polar API is necessary to both intra-frame and inter-frame strategies. In both cases, only the input and output pointers change. The intra-frame strategy exploits SIMD units without increasing the decoder latency. Since it still processes frames one at a time, it preserves fine grain frame pipelining. However, at leaf nodes and nearby, too few elements remain to fill SIMD units. For instance, 4-way SIMD registers are fully filled only at level 2 and above. Thus,

```
1  template <typename B, typename R>
2  class API_polar_SIMD : public API_polar
3  {
4      template <int N_ELMTS = 0> // <- this template parameter is not used here
5      static void f(const R *la, const R *lb, R *lc, const int n_elmts) {
6          for (auto n = 0; n < n_elmts; n += mipp::N<R>())
7          {   // lc = f(la,lb) = sign(la.lb).min(|la|, |lb|)
8              auto r = mipp::copysign(mipp::min(mipp::abs(&la[n]), mipp::abs(&lb[n])),
9                                      mipp::sign(&la[n], &lb[n]));
10             r.store(&lc[n]);
11         }
12     }
13     template <int N_ELMTS = 0> // <- this template parameter is not used here
14     static void g(const R *la, const R *lb, const B *s, R *lc, const int n_elmts) {
15         for (auto n = 0; n < n_elmts; n += mipp::N<R>())
16         {   // lc = g(la,lb,s) = (1-2s)la + lb
17             auto r = mipp::copysign(&la[n], &s[n]) + &lb[n];
18             r.store(&lc[n]);
19         }
20     }
21     template <int N_ELMTS = 0> // <- this template parameter is not used here
22     static void h(const B *sa, const B *sb, B *sc, const int n_elmts) {
23         for (auto n = 0; n < n_elmts; n += mipp::N<R>())
24         {   // sc = h(sa,sb) = sa XOR sb
25             auto r = mipp::R<B>(&sa[n]) ^ mipp::R<B>(&sb[n]);
26             r.store(&sc[n]);
27         }
28     }
29 };
```

Listing 2.7 – Example of a C++ SIMD polar API (f, g and h functions are implemented).

the intra-frame strategy is only effective on trees that can be heavily pruned from these numerous scalar nodes. Even if the tree is heavily pruned some nodes cannot be fully vectorized in the lower layers. In this context, the building blocks of the polar API enable to automatically switch from SIMD to sequential implementations.

2.5.3 Successive Cancellation Decoders

2.5.3.1 Dynamic Implementation

In this section, if a decoder is generic and flexible, it is called *dynamic* (as opposed to *generated* or *unrolled* decoders). The proposed dynamic SSC decoder directly uses the polar API introduced in Section 2.5.2. The same source code is able to accommodate different frozen bit layouts and different parameters (N, K, SNR). It is the first non-generated version (to the best of our knowledge) to support both multi-precisions (32-bit, 16-bit and 8-bit) and multi-SIMD strategies (intra-frame or inter-frame).

The main challenge of the dynamic SSC decoder is to maintain a good performance level at the bottom of the tree. Near the leaves, the decoder spends a non-negligible part of the time in recursive function calls and short loop executions. In Listing 2.7, the loops lines 6, 15, 23 cannot be unrolled by the compiler because `n_elmts` is a runtime parameter. Thus, an useless overhead is due to the loop condition evaluation. To overcome this problem, we wrote a SSC sub-decoder

with the template meta-programming technique. The idea is to fully unroll the recursive calls and to statically give the number of elements in the loops lines 6, 15, 23. To this purpose, a specific `API_polar_SIMD_static` has been developed. The source code is mainly the same as the `API_polar_SIMD` implementation. The only difference is that the `n_elmts` parameter has been replaced by the static `N_ELMTS` parameter in the loops. Then, the compiler is able to unroll the loops. The size of the unrolled sub-tree can be adjusted from a static parameter in the source code. We found that a sub-tree with a depth of 6 gives a good level of performance. Increasing the size of the sub-tree to more than 6 did not give any performance gains in our tests. As a consequence, the proposed flexible decoder cannot decode polar codes smaller than $\mathsf{N} = 2^6 = 64$. This is acceptable knowing that the error-rate performance of the SSC decoder is good for large frame sizes.

To go even further and reach highest throughputs and lowest latencies possible, the next section proposes to fully unroll/generate the SSC decoder.

2.5.3.2 Unrolled/Generated Implementation

Specialized Decoder Skeletons and Polar API The tree structure at the heart of SC decoders is fully determined by the parameters of a given polar code instance: the code size, the code rate ($\mathsf{R} = \mathsf{K}/\mathsf{N}$), the position of the frozen bits. All these parameters are statically known at compile time. Yet, the recursive tree traversal code structure and the corresponding tree data structure are challenging to vectorize and to optimize for a compiler. Our Polar ECC Decoder Generation Environment (P-EDGE) builds on this property to provide a general framework for polar decoder design, generation and optimization [3]. Beyond the *code parameters*, Polar decoders can be tweaked and optimized in many different orthogonal or loosely coupled ways: *Elementary* type (floating-point, fixed-point), *Element containers* (array size), *Data layout* (bit packing techniques), *Instruction Set* (x86, ARM®), *SIMD* support (scalar, intra-frame or inter-frame processing vectorization), *SIMD instruction set variant* (SSE, AVX, AVX-512, NEON), as well as the set and relative priorities of the *rewriting rules for tree pruning*. Our framework enables to quickly experiment the different combinations of all optimizations. Thus, the decoder description results from two distinct parts:

— An architecture independent *specialized decoder skeleton* generated by our decoder generator, from a given frozen bits location input. Starting from the naive, recursive expression of the computational tree, we apply successively cuts and specializations on the tree. They are described through a set of rewriting rules, that can be customized according to the specificities of the decoder and to the constraints in terms of code size for instance (see Section 2.5.1.1).

— A library of architecture dependent *elementary computation building blocks*, corresponding to the implementation variants of the f, g and h functions (fixed- or floating-point versions, scalar or vector versions, ...). These blocks do not depend on the frozen bits location and can therefore be used by any specialized skeleton (see Section 2.5.2).

This separation of concerns, between high-level specialized algorithmic skeletons and low-level arithmetic routines, enables both ECC experts to focus on optimizing algorithm skeletons and architecture experts to focus on writing highly optimized routines.

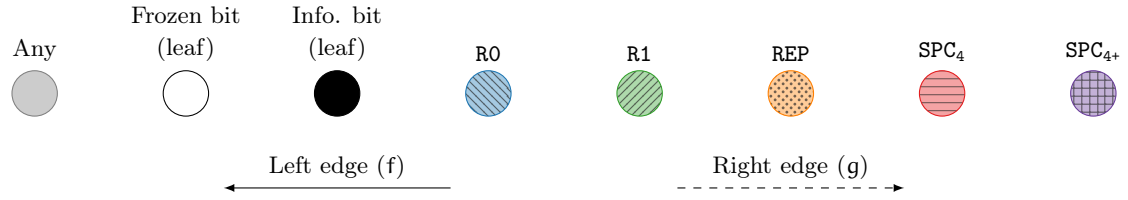
3. P-EDGE repository: https://github.com/aff3ct/polar_decoder_gen

```
// the frozen bits definition (1 = frozen, 0 = not frozen)
static const std::vector<bool> Decoder_polar_SC_N8_K4 = {
1, 1, 1, 0, 1, 0, 0, 0};
// the generated decoder class with templatized bit ('B') and real ('R') types
// and the generic polar API ('API_polar') can be seq., inter- and intra-SIMD
template <typename B, typename R, class API_polar>
class Decoder_polar_SC_N8_K4
{
public:
    // the 'decode' method: recursive function calls are fully unrolled
    // 'l' is the input and intermediate vector of LLRs (RW)
    // 's' is the partial sums and the output vector of bits (RW)
    void decode()
    {   // ...                  n_elmts   read   read   read  write  n_elmts
        //                       static    l/s    l/s    l/s    l/s  dynamic
        API_polar::template f  <      4>(l + 0, l + 4,        l + 8,       4);
        API_polar::template rep<      4>(l + 8,               s + 0,       4);
        API_polar::template gr <      4>(l + 0, l + 4, s + 0, l + 8,       4);
        API_polar::template spc<      4>(l + 8,               s + 4,       4);
        API_polar::template h  <      4>(s + 0, s + 4,        s + 0,       4);
    }
    /* ... */
};
```

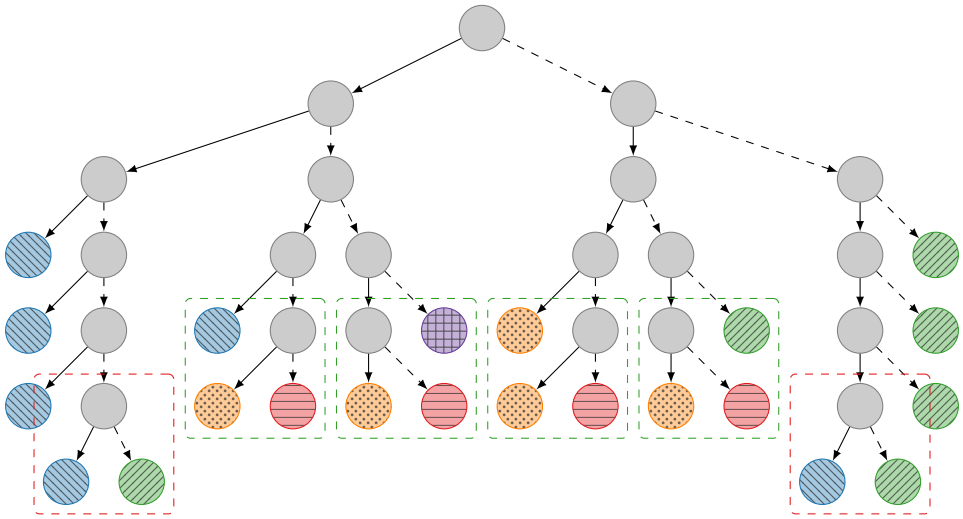
Listing 2.8 – Generated polar SC decoder source code corresponding to the pruned tree in Figure 1.9.

Decoder Generation The decoder generator first builds the binary tree structure from the pattern matching algorithm presented in Section 2.5.1.1. Then, once the tree has been fully specialized, the P-EDGE generator performs a second tree traversal pass to output the resulting decoder. An example of such a tree specialization process together with the generator output is shown in Figure 1.9 and in Listing 2.8. In Listing 2.8, each operation (`f`, `rep`, `gr`, `spc` and `h`) is applied on 4 elements. This number of elements is given as a static template parameter as well as a standard function parameter. Depending on the selected polar API, one or the other will be used. Of course with fully unrolled decoders, it is much more efficient to use an API implementation that uses the template parameter to fully unroll the loops at compile time.

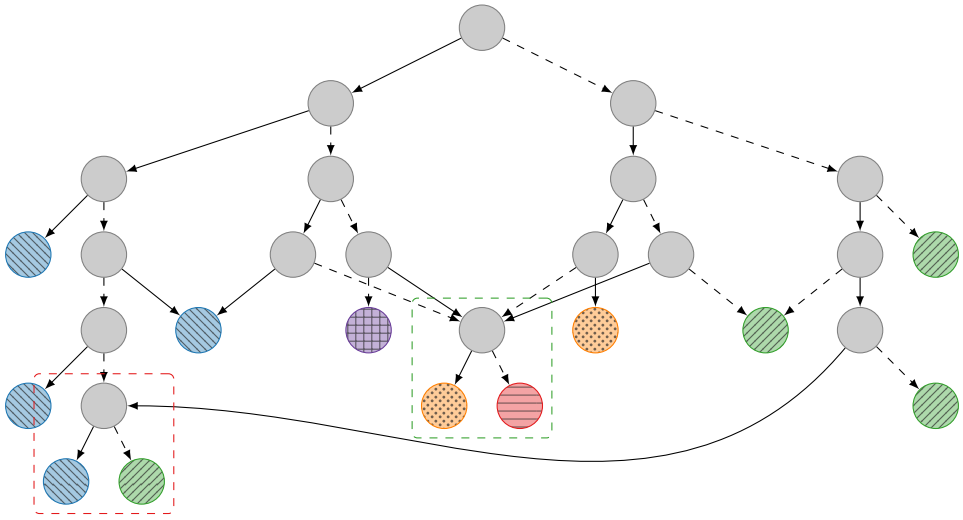
Source Code Compression Decoders are generated as straight-line code (no recursive calls), with all node computations put in sequence. This improves performance for small to medium codeword sizes, up to the point where the compiled binary exceeds the L1I cache size (this is also reported in [Gia+16]). We mitigated this issue by reducing decoder binary sizes using two compression techniques: 1) in the generated code, we moved the buffer offsets from template arguments to function arguments, which enabled the compiler to factorize more function calls than before (improvement by a factor of 10), 2) we implemented a sub-tree folding algorithm in the P-EDGE generator (see Figure 2.8), to detect multiple occurrences of a same sub-tree and to put the corresponding code into a dedicated function. These techniques lead to an improvement by a factor of 5 for $N = 2^{16}$ knowing that the compression ratio increases with the size of the tree.



(a) Legend.



(b) Without compression.



(c) With compression.

Figure 2.8 – Pruned polar decoding tree representation ($\mathsf{N} = 128, \mathsf{K} = 64$) without and with the compression sub-tree folding algorithm.

2.5.3.3 LLRs and Partial Sums Memory Management

The memory management is similar in the flexible SSC decoder and in the unrolled versions. Each time, the decoder stores its state using two data buffers, one for the LLR values ($\boldsymbol{\lambda}$) and the other for the bits (partial sums $\hat{\mathbf{s}}$). The "logical" tree layout is implemented as a simple and efficient *heap* vector data layout. Therefore traversing the tree corresponds to moving through the array, at different offsets and considering different index intervals. The LLR offset is computed from the graph depth $\mathtt{d}$ (the node vertical indexing) as follows:

$$o_\lambda(\mathtt{d}) = \begin{cases} 0 & \mathtt{d} = 0, \\ \sum_{\mathtt{i}=1}^{\mathtt{d}} \frac{N}{2^{\mathtt{i}-1}} & \text{otherwise.} \end{cases} \tag{2.2}$$

Given $\mathtt{l_a}$ the lane (the node horizontal indexing), the bit offset is determined as follows:

$$o_{\hat{s}}(\mathtt{d}, \mathtt{l_a}) = \frac{N}{2^{\mathtt{d}}} \times \mathtt{l_a}. \tag{2.3}$$

The LLR buffer size is $2N$ and the bit buffer is N, for a frame of N bits. Thus, the memory footprint per frame can be expressed:

$$\mathtt{mem_{fp}} = N \times (2 \times \text{sizeof}(\mathtt{LLR}) + \text{sizeof}(\mathtt{bit})). \tag{2.4}$$

LLRs element size is 4 bytes (float) or 1 byte (fixed-point numbers). The inter-SIMD version also employs a *bit packing* memory footprint reduction technique [LLJ15] to pack several bits together by using shifts and masking instructions.

2.5.4 Successive Cancellation List Decoders

In this section, the proposed implementation of the SSCL class of decoders focuses on intra-frame SIMD dynamic decoders. A description of a fully unrolled SSCL decoder has been previously evaluated in [Sar+16]. But, we want to oppose it to more generic and flexible decoders. Some implementation improvements are necessary in order to be competitive with specific unrolled decoders of the literature. The polar API (see Section 2.5.2) enables to benefit from the SIMD instructions for various target architectures. Optimizations of CRC checking benefit to both the non-adaptive and adaptive versions of the CA-SSCL algorithms. The new sorting technique presented in Section 2.5.4.2 can be applied to each variation of the SSCL algorithm. Finally, an efficient implementation of the partial sums memory management is proposed in Section 2.5.4.3. It is particularly effective for short polar codes.

2.5.4.1 Improving Cyclic Redundancy Checking

By profiling the Adaptive SCL decoder, one may observe that a significant amount of time is spent to process the cyclic redundancy checks. Its computational complexity is $O(LN)$ versus the computational complexity of the SCL decoding, $O(LN \log N)$. The first is not negligible compared to the second. In the adaptive decoder, the CRC verification is performed a first time after the SC decoding. In the following, we show how to reduce the computational complexity of these CRC verifications.

First, an efficient CRC checking code has been implemented. Whenever the decoder has to check the CRC, the bits are packed and then computed 32 by 32. In order to further speed up the implementation, a lookup table used to store pre-computed CRC sub-sequences, and thus reduce the computational complexity. The size of the lookup table is 1 KB.

After a regular SC decoding, a decision vector of size N is produced. Then, the K information bits must be extracted to apply cyclic redundancy check. The profiling of our decoder description shows that this extraction takes a significant amount of time compared to the check operation itself. Consequently, a specific extraction function was implemented. This function takes advantage of the leaf node type knowledge to perform efficient multi-element copies.

Concerning SCL decoding, it is possible to sort the candidates according to their respective metrics and then to check the CRC of each candidate from the best to the worst. Once a candidate with a valid CRC is found, it is chosen as the decision. This method gives similar decoding performance as performing the CRC of each candidate and then selecting the one with the best metric. With the adopted order, decoding time is saved by reducing the average number of checked candidates. This is made in the "selectBestPath()" sub-routine (Algorithm 1.1, l.18).

2.5.4.2 LLR and Metric Sorting

Metric sorting is involved in the path selection step, but also in the "updatePaths()" sub-routine (Algorithm. 1.1, l.16) and consequently in each leaf. Sorting the LLRs is also necessary in `R1` and `SPC` nodes. Because of a lack of information about the sorting technique presented in [Sar+16], its reproduction is not possible. In this paragraph, the sorting algorithm used in our proposed SCL decoder is described.

In `R1` nodes, a Chase-2 [Cha72] algorithm is applied. The two minimum absolute values of the LLRs have to be identified. The way to do the minimum number of comparisons to identify the 2 largest of $n \geqslant 2$ elements was originally described by Schreier in [Sch32] and reported in [Knu73]. The lower stages of this algorithm can be parallelized thanks to SIMD instructions in the way described in [FAN07]. According to our experimentations, Schreier's algorithm is the most efficient compared to parallelized Batcher's merge exchange, partial quick-sort or heap-sort implemented in the `C++` standard library in the case of `R1` nodes. At the end, we chose not to apply the SIMD implementation of Schreier's algorithm because: 1) the speedup was negligible, 2) in 8-bit fixed-point, only $N \leqslant 256$ codewords can be considered.

Concerning path metrics, partial quick-sort appeared to yield no gains in terms of throughput by comparison with the algorithm in [Sch32], neither did heap-sort or parallelized Batcher's merge exchange. For a matter of consistency, only Schreier's algorithm is used in the proposed decoder, for both LLR sorting in `R1` and `SPC` nodes and for path metrics sorting. The sorting of path metrics is applied to choose the paths to be removed, kept or duplicated.

2.5.4.3 Partial Sums Memory Management

An SCL decoder can be seen as L replications of an SC decoder. The first possible memory layout is the one given in Figure 1.8. In this layout, the partial sums $\hat{s}$ of each node is stored in a dedicated array. Therefore, a memory of size $2N-1$ bits is necessary in the SC decoder, or $L(2N-1)$ bits in the SCL decoder. This memory layout is described in [TV11] and present in previous software implementations [Sar+14c, Sar+16, She+16].

A possible improvement is to change the memory layout to reduce its footprint. Due to the order of operations in both SC and SCL algorithms, the partial sums on a given layer are only used once by the h function and can then be overwritten. Thus, a dedicated memory allocation is not necessary at each layer of the tree. The memory can be shared between the stages. Therefore the memory footprint can be reduced from $2N-1$ to N in the SC decoder as shown in [Ler+13]. A reduction from $L(2N-1)$ to LN can be obtained in the SCL decoder.

In the case of the SCL algorithm, L paths have to be assigned to L partial sum memory arrays. In [TV11], this assignment is made with pointers. The advantage of pointers is that when a path is duplicated, in the "updatePaths()" sub-routine of Algorithm 1.1, the partial sums are not copied. Actually, they can be shared between paths thanks to the use of pointers. This method limits the number of memory operations. Unfortunately, it is not possible to take advantage of the memory space reduction. Indeed, the partial sums have to be stored on $L(2N-1)$ bits. There is an alternative to this mechanism. If a logical path is statically assigned to a memory array, no pointer is necessary at the cost that partial sums must be copied when a path is duplicated (only LN bits are required). This method is called $\text{SSCL}_{\texttt{cpy}}$ whereas the former is called $\text{SSCL}_{\texttt{ptr}}$.

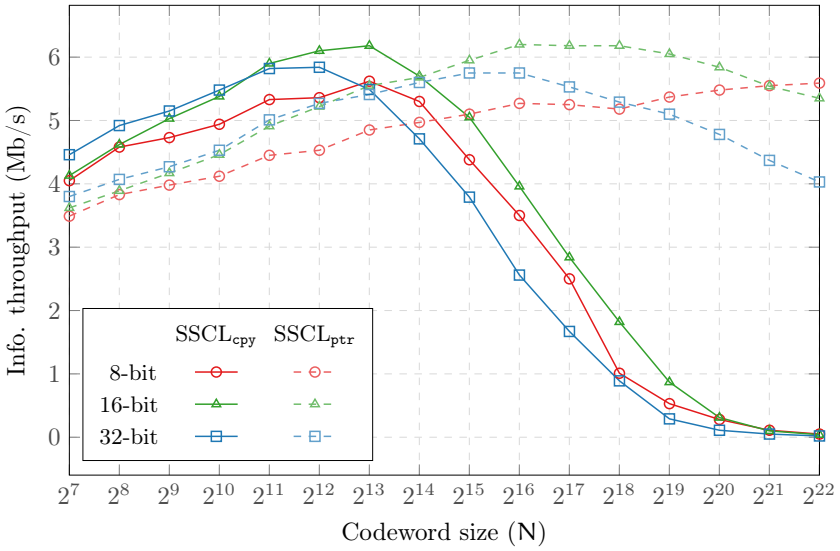


Figure 2.9 – Information throughput of the SSCL decoder depending on the codeword size (N) and the partial sums management. $R = 1/2$, $L = 8$ (on the Intel® Core™ i5-6600K CPU).

Our experiments have shown that the overhead of handling pointers plus the extra memory space requirement cause the $\text{SSCL}_{\texttt{cpy}}$ to be more efficient than the $\text{SSCL}_{\texttt{ptr}}$ for short and medium code lengths, as shown in Figure 2.9. The 32-bit version uses floating-point LLRs, whereas 16-bit and 8-bit versions are in fixed-point. Notice that in this work, each bit of the partial sums is stored as an 8-bit, 16-bit or 32-bit number accordingly to the LLR data type. The code rate R is equal to 1/2. The throughput of the $\text{SSCL}_{\texttt{cpy}}$ version is higher for $N \leqslant 8192$ whereas the $\text{SSCL}_{\texttt{ptr}}$ version is more efficient for higher values of N. Figure 2.9 also illustrates the impact of the representation of partial sums. For very high values of N, the 8-bit fixed point representation takes advantage of fewer cache misses. As the decoding performance improvements of the SCL algorithm are not very significant compared to the SC algorithm for long polar codes, $\text{SSCL}_{\texttt{cpy}}$ is the appropriate solution in most practical cases.

In our decoder description, LLRs are managed with pointers, as it is the case in other software implementations of the literature [Sar+14c, Sar+16, She+16]. We tried to remove the pointer handling as for the partial sums, but this was not beneficial in any use case.

2.5.4.4 Memory Footprint

Table 2.6 – Polar decoders memory complexity.

Algorithms	Memory Footprint
(CA-)SSCL$_{\texttt{cpy}}$	$\mathcal{O}((2\mathsf{L}+1)\mathsf{NQ})$
(CA-)SSCL$_{\texttt{ptr}}$	$\mathcal{O}((3\mathsf{L}+1)\mathsf{NQ})$
A-SSCL$_{\texttt{cpy}}$	$\mathcal{O}((2\mathsf{L}+3)\mathsf{NQ})$
A-SSCL$_{\texttt{ptr}}$	$\mathcal{O}((3\mathsf{L}+3)\mathsf{NQ})$

The exact memory footprint of the SSCL decoders is hard to estimate as there are many small buffers related to the implementation. However, the memory footprint is mainly driven by the LLRs ($\boldsymbol{\lambda}$) and the partial sums ($\hat{\mathbf{s}}$) as they linearly depend on LN. The buffers related to the path metrics can be neglected as they linearly depend on L. The memory footprint of the CRC is also negligible, the only requirement is a lookup table of 256 integers. Table 2.6 summarizes the memory footprint estimation of the various decoders while Q stands for the size of the element (1, 2 or 4 bytes). The channel LLRs are taken into account in the approximation. As explained in the previous section, the SSCL$_{\texttt{ptr}}$ version of the code requires twice the amount of data for the partial sums. Notice that the memory footprint of the adaptive decoders is a little bit higher than the other SSCL decoder since it includes an additional SSC decoder.

In this section we proposed flexible software implementations of the SC and the SCL decoding algorithms. The pruned versions of these decoders (SSC and SSCL) are the key of efficiency. The generated (or unrolled) strategy has also been experimented and improved for the SSC decoders. These specialized decoders trade a part of the flexibility for higher throughput and lower latency performance.

2.6 Turbo Decoders

A turbo decoder is in charge of decoding a large set of frames. Two strategies are then possible to speedup the decoding process. i) *intra-frame parallelism*: the decoder exploits the parallelism within the turbo-decoding process by executing concurrent tasks during the decoding of one frame. ii) *inter-frame parallelism*: several frames are decoded simultaneously. In the perspective of a hardware implementation, the intra-frame approach is efficient [MBJ09] because the area overhead resulting from parallelization is lower than the speedup. On the contrary, the inter-frame strategy is inefficient, due to the duplication of multiple hardware turbo-decoders. The resulting speedup comes at a high cost in terms of area overhead.

In the perspective of a software implementation, the issue is different. The algorithm is executed on a programmable non-modifiable architecture. The degree of freedom lies in the mapping of the different parallelizable tasks on the parallel units of the processor. Modern multi-core processors support Single Program Multiple Data (SPMD) execution. Each core includes Single Instruction Multiple Data (SIMD) units. The objective is then to identify the parallelization strategy suitable for both SIMD and SPMD programming models. In the literature, intra-frame parallelism is often mapped on SIMD units while inter-frame parallelization is usually kept for multi-threaded approaches (SPMD). In [Zha+12, Wu+13], multiple trellis-state computations are performed in parallel in the SIMD units. In [WSC10, Wu+11, CS12, YC12, Zha+12, Liu+13, Che+13, Xia+13, Wu+13, Zha+14b, Li+14], the decoded frame is split into sub-blocks that are processed in parallel in the SIMD units. An alternative approach is to process both SISO decoding in parallel but, it requires additional computations for synchronization and/or impacts on error-

correction performance [MBJ09]. However, for all these approaches a part of the computation of the BCJR decoder remains sequential, bounding the speedup below the capabilities of SIMD units. Inter-frame parallelism has been proposed in [WSC10, Wu+11, Zha+12, Wu+13]. Multiple codewords are decoded in parallel, this improves the memory access regularity and the usage rate of SIMD units. The speedup is no longer bounded by the sequential parts, all removed, but this comes at the expense of an increase in memory footprint and latency. In this work, we focus on the inter-frame parallelization and show that the use of this approach enables some register-reuse optimizations that are not possible in the intra-frame strategy.

2.6.1 Inter-frame Parallelism on Multi-core CPUs

The contribution of this work is to propose an efficient mapping of multiple frames on the CPU SIMD units (inter-frame strategy): the decoding of F frames is vectorized. Before the decoding process can be launched, this new approach requires to: (a) buffer a set of F frames and (b) reorder the input LLRs in order to make the SIMDization efficient with memory aligned operations (see Section 2.2.2). Similarly, a reverse-reordering step has to be performed at the end of the turbo decoding. These reordering operations are expensive but they make the complete decoding process very regular and efficient for SIMD parallelization. Moreover, reordering is applied only once, independently of the number of decoding iterations.

Algorithm 2.1: Loop fusion BCJR implementation.

```
1  for all frames do                                                    ▷ Vectorized loop
2      α^0 ← initAlpha()
3      for k = 1; k < K; k = k + 1 do                                   ▷ Sequential loop
4          γ^{k-1} ← computeGamma(L_s^{k-1}, L_p^{k-1}, L_a^{k-1})
5          α^k ← computeAlpha(α^{k-1}, γ^{k-1})
6      γ^{K-1} ← computeGamma(L_s^{K-1}, L_p^{K-1}, L_a^{K-1})
7      β^{K-1} ← initBeta()
8      L_e^{K-1} ← computeExtrinsic(α^{K-1}, β^{K-1}, γ^{K-1}, L_s^{K-1}, L_a^{K-1})
9      for k = K − 2; k ⩾ 0; k = k − 1 do                               ▷ Sequential loop
10         β^k ← computeBeta(β^{k+1}, γ^k)
11         L_e^k ← computeExtrinsic(α^k, β^k, γ^k, L_s^k, L_a^k)
```

In the proposed implementation, the inter-frame parallelism is used to fill the SIMD units of the CPU cores. Algorithm 1.2 illustrates the traditional implementation of the BCJR (used for the *intra-frame* vectorization). The inter-frame strategy makes the outer loop on the frame parallel (through vectors). This means all computations inside this loop operate on SIMD vectors instead of scalars. The inner loops can be turned into sequential loops on SIMD vectors. This gives the opportunity for memory optimizations, through loop fusion. The initial 4 inner loops are merged into 2 loops. Algorithm 2.1 presents this loop fusion optimization. This makes possible the scalar promotion of β_j (no longer an array). Indeed, it can be directly reused from the CPU registers. In this version, the SIMD are always stressed.

On a multi-core processor, each core decodes F frames using its own SIMD unit. As T threads are activated, a total of $\mathsf{F} \times \mathsf{T}$ frames are therefore decoded simultaneously with the inter-frame strategy. Theoretically, this SPMD parallelization strategy provides an acceleration up to a factor T, with T cores. Large memory footprint exceeding L3 cache capacity may reduce the effective speedup, as shown in Section 4.3.

2.6.2 Software Implementation of the Turbo Decoder

2.6.2.1 Fixed-point Representation

Nowadays on x86 CPUs, there are large SIMD registers: SSE/NEON are 128 bits wide and AVX are 256 bits wide. The number of elements that can be vectorized depends on the SIMD length and on the data format: $\mathsf{p}_{\mathrm{SIMD}} = \mathrm{sizeof(SIMD)}/\,\mathrm{sizeof(data)}$. So, the key for a wide parallelism is to work on short data.

During the turbo-decoding process, the extrinsic values grow at each iteration. It is then necessary for internal LLRs to have a larger dynamic than the channel information. Depending on the data format, 16-bit or 8-bit, the quantization used in the decoder is $\mathsf{Q}_{16,3}$ or $\mathsf{Q}_{8,2}$, respectively.

2.6.2.2 Memory Allocations

The systematic information $\mathbf{L_s}/\mathbf{L'_s}$ and the parity information $\mathbf{L_p}/\mathbf{L'_p}$ are stored in the natural domain $\mathcal{N}$ as well as in the interleaved domain $\mathcal{I}$. Moreover, two extrinsic vectors are stored: $\mathbf{L_{e:1\to 2}}$ in $\mathcal{N}$ and $\mathbf{L_{e:2\to 1}}$ in $\mathcal{I}$ as well as two a priori vectors: $\mathbf{L_{a:1\to 2}}$ in $\mathcal{I}$ and $\mathbf{L_{a:2\to 1}}$ in $\mathcal{N}$. Inside the BCJR decoding and per trellis section, two γ_i and eight α_j metrics are stored. Thanks to the loop fusion optimization, the eight β_j metrics are not stored in memory. In the proposed implementation $i \in \{0,1\}$ and $j \in \{0,1,2,3,4,5,6,7\}$. Notice that all those previously-mentioned vectors are K-bit wide and are duplicated $F \times T$ times because of the inter-frame strategy. The memory footprint in bytes is approximately: $18 \times K \times \mathrm{sizeof(data)} \times F \times T$ (where $F = \mathsf{p}_{\mathrm{SIMD}}$). The interleaving and deinterleaving lookup tables have been neglected in this model.

2.6.2.3 Forward Trellis Traversal

The objective is to reduce the number of loads/stores by performing the arithmetic computations (`add` and `max`) inside registers. The max-log-MAP (ML-MAP) algorithm only stresses the integer pipeline of the CPU. This kind of operations takes only one cycle to execute when the latency is also very small (1 cycle too). In contrast, a load/store can take a larger number of cycles depending on where the current value is loaded/stored in the memory hierarchy. Using data directly from the registers is cost-free but loading/storing it from the L1/L2/L3 cache can take up to 30 cycles (at worst).

Per trellis section k, the two γ_i^k metrics are computed from the systematic and the parity information. These two γ_i^k are directly reused to compute the eight α_j^k metrics. Depending on the number of bits available, the trellis traversal requires to normalize the α_j^k because of the accumulations along the multiple sections. In 8-bit format, the α_j^k metrics are normalized for each section: the first α_0^k value is subtracted from all the α_j^k (including α_0^k itself). In the 16-bit decoder, the normalization is only applied every eight steps, since there are enough bits to accumulate eight values. We have observed in experiments that there is no performance degradation due to the normalization process. At the end of a trellis section k the two γ_i^k and the eight normalized α_j^k are stored in memory. In the next trellis section $(k+1)$ the eight previous α_j^k are not loaded from memory but they are directly reused from registers to compute the α_j^{k+1} values.

2.6.2.4 Backward Trellis Traversal

Per trellis section k, the two γ_i^k metrics are loaded from the memory. These two metrics are then used to compute, on the fly, the eight β_j^k metrics (whenever needed the β_j^k metrics have been normalized like for the α_j^k metrics). After that, the α_j^k metrics are loaded from the memory. The α_j^k, β_j^k and γ_i^k metrics are used to determine the *a posteriori* and the extrinsic LLRs. In the next trellis section $(k-1)$ the previous β_j^k metrics are directly reused from registers in order to compute the next β_j^{k-1} values. The β_j^k metrics are never stored in memory.

2.6.2.5 Loop Unrolling

```
1  for (auto j = 0; j < n_states; j++) // in the LTE standard 'n_states' = 8
2  {
3      const mipp::Reg<R> alpha_kprev1_j = &alpha[trellis[1][j]][k -1];
4      const mipp::Reg<R> alpha_kprev2_j = &alpha[trellis[2][j]][k -1];
5      auto alpha_k_j = mipp::max(alpha_kprev1_j + gamma, alpha_kprev2_j - gamma);
6  }
```

Listing 2.9 – Generic implementation of the $\boldsymbol{\alpha^k}$ computations.

```
1  const mipp::Reg<R> alpha_kprev_0 = &alpha[0][k -1];
2  const mipp::Reg<R> alpha_kprev_1 = &alpha[1][k -1];
3  // ...
4  const mipp::Reg<R> alpha_kprev_7 = &alpha[7][k -1];
5  auto alpha_k_0 = mipp::max(alpha_kprev_0 + gamma, alpha_kprev_1 - gamma);
6  auto alpha_k_1 = mipp::max(alpha_kprev_3 + gamma, alpha_kprev_2 - gamma);
7  auto alpha_k_2 = mipp::max(alpha_kprev_4 + gamma, alpha_kprev_5 - gamma);
8  auto alpha_k_3 = mipp::max(alpha_kprev_7 + gamma, alpha_kprev_6 - gamma);
9  auto alpha_k_4 = mipp::max(alpha_kprev_1 + gamma, alpha_kprev_0 - gamma);
10 auto alpha_k_5 = mipp::max(alpha_kprev_2 + gamma, alpha_kprev_3 - gamma);
11 auto alpha_k_6 = mipp::max(alpha_kprev_5 + gamma, alpha_kprev_4 - gamma);
12 auto alpha_k_7 = mipp::max(alpha_kprev_6 + gamma, alpha_kprev_7 - gamma);
```

Listing 2.10 – Unrolled implementation of the $\boldsymbol{\alpha^k}$ computations.

The computations of the eight α_j^k states can be implemented with a generic `trellis` structure as shown in Listing 2.9. The main problem is that the loop line 1 cannot be unrolled by the compiler and there is extra memory accesses (or indirections) due to the `trellis` vector. Knowing precisely the structure of the trellis in the LTE standard (see Figure 1.13), it is possible to fully unroll the loop. The unrolled description is shown in Listing 2.10. This version is adopted in the Section 4.3 as it leads to significant throughput improvements. The same optimization is also applied to the computation of the β_j^k states. One can note that the source code examples are simplified: `gamma` represents the corresponding γ_i^k.

In this section, we presented a high throughput implementation of the turbo decoding algorithm. This implementation largely relies on the inter-frame SIMD strategy combined with a fixed-point representation of the LLRs and some specializations for the LTE 8-state trellis.

2.7 SCMA Demodulators

Besides methodical improvements of the MPA such as log-MPA, hardware oriented improvements are important to take full benefit of C-RAN servers capabilities. Since MPA and log-MPA are control heavy algorithms, mishandling of data can induce huge performance losses. This section explores how MPA can be reformulated: 1) to improve data locality in cache and to reduce cache misses and branch mispredictions 2) to reorder the data paths in order to help exploiting data-level parallelism at each step of the MPA and log-MPA algorithms and 3) to exploit approximated modeling of additive white Gaussian noise in order to eliminate exponential calculations and to drastically reduce the number of SIMD instructions.

2.7.1 Flattening Matrices to Reduce Cache Misses and Branch Misses

Considering (1.15), there are 64 calculations of distances and probabilities for each resource (256 for all resources). Using a multidimensional array ($4 \times 4 \times 4$) should be avoided, because it typically causes bad data locality, which leads to an increased number of cache misses. These misses negatively affect the throughput, and this is significant, since this process must be repeated in the decoder for each received 12-bit block of data. Flattening a $\mathtt{d}$-dimensional array to a vector using (2.5) is appropriate to prevent cache misses and improve the spatial locality of data. This is done with the help of an index defined as:

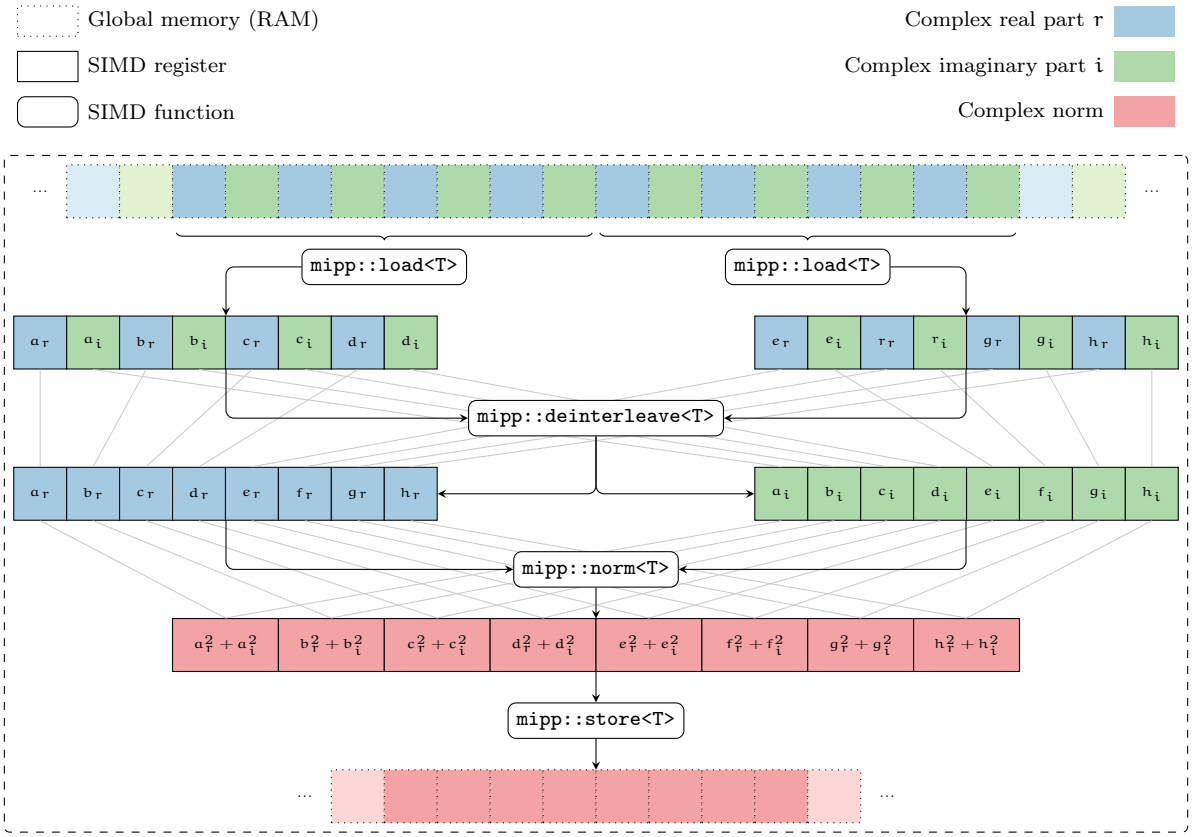
$$\text{index} = \sum_{i=1}^{d} \left(\prod_{j=i+1}^{d} N_j \right) n_i. \tag{2.5}$$

Where N_j is the size of the j^{th} dimension of the array and n_i is the location of a target element in that dimension. Improving data locality with a stride of a single floating-point number in each element makes it easier for the processor to have aligned and contiguous accesses to the memory through SIMD ISA. SIMD instructions help to reduce the total number of mispredicted branches in the algorithm. Contiguous accesses to the L1 cache are performed by chunks of 128-bit, 256-bit or 512-bit. This reduces the number of iterations in the `for`-loops and consequently it reduces the number of branches. On the other hand, for a vector of sixty four 32-bit floating-point numbers, 64 iterations are necessary in the scalar mode, while only 16, 8 or 4 iterations are required in the vectorized modes using SSE (or NEON), AVX or AVX-512 (or KNCI) ISAs, respectively.

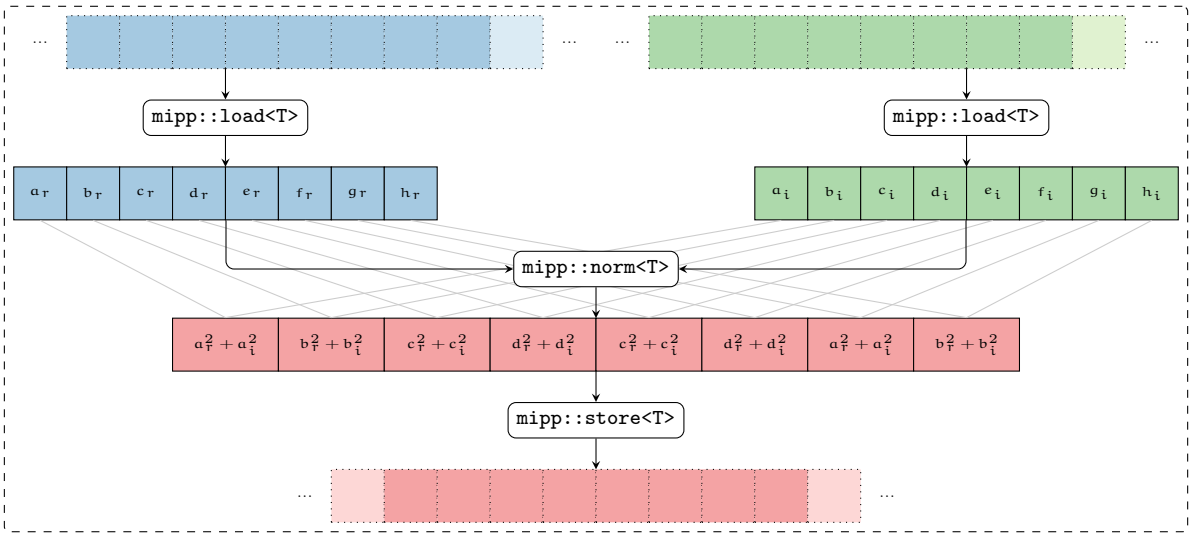
2.7.2 Adapting the Algorithms to Improve Data-level Parallelism

The SIMD instructions provide high-performance loads and stores to the cache memory due to data vectorization. Flattening matrices to vectors is a prerequisite to enable SIMD contiguous accesses to memory. In the presented work the SIMD operations are made on 32-bit floating-point real numbers ($\mathtt{T}$ = `float`). The proposed implementations are working on $F = J = 6$ frames. The F frames are not independent so we consider that the SIMD strategy is similar to the intra-frame SIMD strategy presented in Section 2.2.1. For the MPA, the SIMD instructions are used to 1) compute the complex norm $\|.\|$ in (1.14), 2) calculate the exponentials in (1.15), 3) perform users to resources messaging and final beliefs at each user in (1.18).

SIMD Computation of Complex Norms Equation 1.14 use a complex norm function $\|.\|$. It can be optimized by using SIMD instructions. There are two ways to perform this computation



(a) Array of Structures (AoS)



(b) Structure of Arrays (SoA)

Figure 2.10 – MPA vectorized complex norm computations ($\mathfrak{p}_{\text{SIMD}} = 8$).

depending on the initial data layout. Figure 2.10a depicts how to implement the norm function with an Array of Structures (AoS) layout for complex numbers. In this data layer, the complex numbers are represented as two consecutive floating-point numbers. The implementation with AoS uses five MIPP functions: two `mipp::load`, one `mipp::deinterleave`, one `mipp::norm` and one `mipp::store`. The MIPP loads and stores are equivalent to real SIMD move instructions. The `mipp::deinterleave` operation can contain from 4 to 12 real assembly instructions. It depends on the data type `T` and the SIMD ISA. The `mipp::norm` operation performs two multiplications and one addition. Figure 2.10b sketches the computation of the complex norm using a Structure of Array (SoA) data layout. This implementation does not require the MIPP `mipp::deinterleave` operation. The real and imaginary parts of the complex numbers are initially separated in memory. Our experiments demonstrated that the SoA method leads to higher performance than the AoS method. This is due to the economy of the `mipp::deinterleave` operation. Depending on the SIMD ISA the deinterleaving can take up to 20% longer. The SoA data layout is used for evaluations.

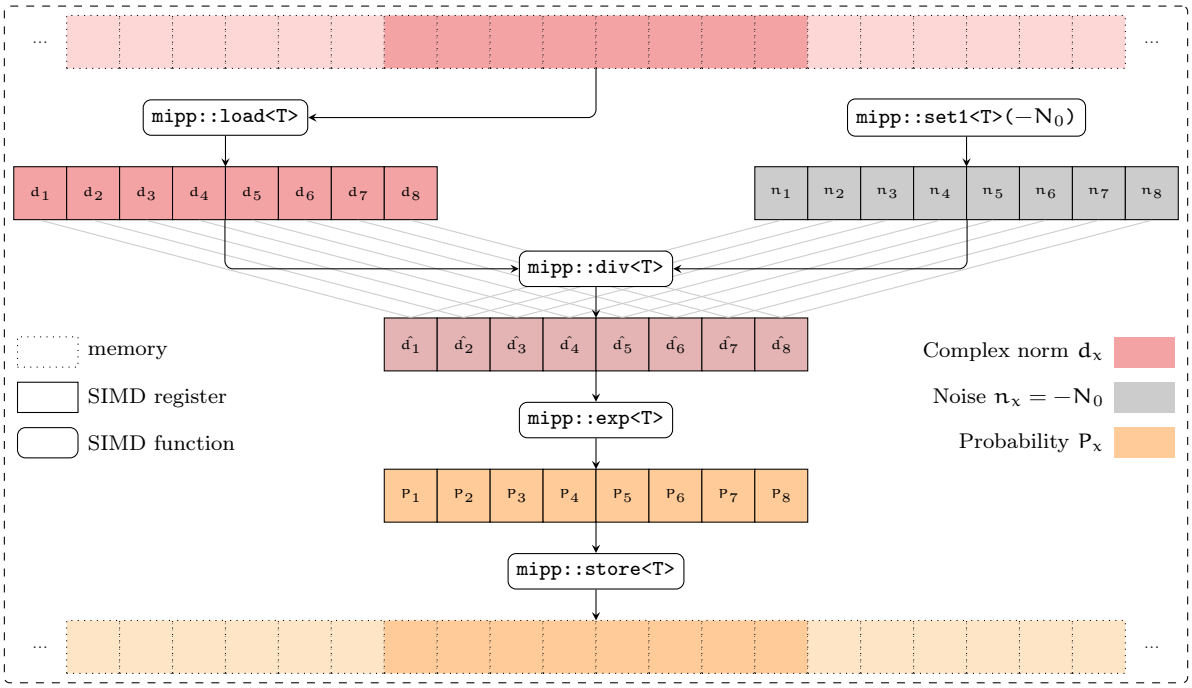


Figure 2.11 – MPA vectorized exponentials ($N_0 = 2\sigma^2$, $p_{SIMD} = 8$).

SIMD Computation of Exponential To speedup the computation of the exponentials used in (1.15), the `mipp::exp` math function is used. The flattened complex and normalized numbers are calculated as shown in Figure 2.10 to produce the preliminary values used to compute the probabilities. Figure 2.11 illustrates the full process on a vector of eight floating-point numbers ($p_{SIMD} = 8$). First the values are loaded into SIMD registers. Then they are multiplied by $-1/2\sigma^2$. Finally the exponential function is performed according to (1.15).

Exponential Approximation with the Estimated-MPA algorithm In the proposed E-MAP algorithm approximation, (1.24) replaces the `mipp::exp` function used in Figure 2.11. It reduces the overall number of instructions to two multiplications and one addition. Knowing that the `mipp::exp` function represents about 30 SIMD assembly instructions, the E-MAP algorithm leads to a drastic reduction of the computation effort.

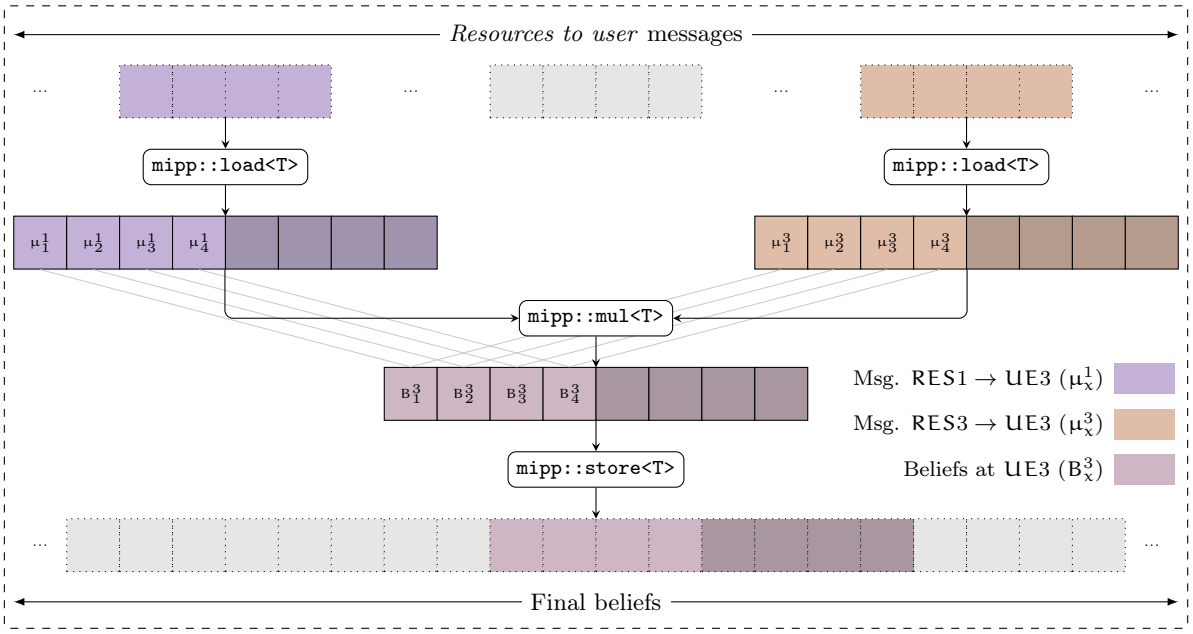


Figure 2.12 – MPA vectorized computations of final beliefs ($p_{SIMD} = 8$).

SIMD Message Passing Some remaining parts of the MPA can be vectorized too. Especially, the guess swaps and the computation of the final beliefs. Each user node can be vectorized but the available level of parallelism is limited to 4 elements. Figure 2.12 shows the computation of final beliefs for user 3 (this is illustrated in Figure 1.17d (III)). There are four messages from a resource to a user containing the probabilities of four different codewords. If $p_{SIMD} > 4$ then some elements of the SIMD register are not used. The data layout has been adapted and the memory allocation is padded. By this way, the read and written extra-elements do not produce segmentation fault errors.

Accuracy of Floating-point Computations The finite precision of floating-point calculations induces losses in the results. Thus, technical standards such as IEEE 754 define rounding rules, precision of calculations, exception handling and underflow behavior. However, the MPA delivers the same bit error rate results with less precise floating-point models. For instance, in the GNU compiler, `-Ofast` is a high-level compiler option which includes fast math libraries to handle floating-point calculations (`-ffast-math`). The compiler uses various mathematical simplifications as explained in [GCC18]. It also uses approximated tables for the division and the square root functions. The compiler also forces the value to zero in the case of an underflow. Using `-Ofast` can improve the throughput of the MPA algorithm as will be shown in Section 4.5.

In this section, we proposed a generic intra-frame SIMD software implementation of the SCMA MPA class of algorithms. The vectorized sub-parts of the algorithms have been detailed and the corresponding MIPP implementations have been given. Other well-known optimization techniques, such as loops unrolling, avoiding type conversions and functions inlining have been used to enhance the throughput of the various message passing algorithms.

2.8 Conclusion

In this chapter, generic strategies for efficient algorithm implementations on CPUs are presented first. The vectorization is a key for performance efficiency on current CPUs. Thus, a main contribution in this chapter is the proposition of MIPP: a wrapper for the SIMD instructions. The idea is to abstract data types and SIMD ISAs in order to propose "universal" and efficient implementations of digital communication receiver algorithms. We show that MIPP introduces little overhead over specific intrinsic functions (or assembly code) and is able to operate on floating-point representations as well as fixed-point ones. For digital communication receiver algorithms, fixed-point representations are very interesting for channel coding algorithms because they offer increased SIMD widths, with moderate impact on the decoding performance. To summarize, MIPP improves the source code flexibility and portability while keeping the same level of performance. Note that the MIPP wrapper has been published in a scientific conference [7].

In a second part, two main vectorization strategies are explicitly defined and presented. The intra-frame SIMD strategy operates on a single frame relying on the algorithm inherent parallelism while the inter-frame SIMD strategy operates on multiple frames at the same time. The intra-SIMD can increase the throughput as well as the latency. On the contrary, the inter-SIMD does not improve the latency but comes with a potentially higher SIMD efficiency and can lead to very high throughputs. These two strategies can be applied to all the processing blocks of digital communication chains. Thus, they are a key point to address the algorithmic heterogeneity problem.

Then, two expensive blocks of functional simulations are studied: the channel and the quantizer. Both algorithms are implemented with MIPP by the mean of the intra-frame SIMD strategy. This results in high performance implementations to deliver fast functional simulations.

The four last sections focus on the design of efficient software implementations of the algorithms presented in Chapter 1 (LDPC decoders, polar decoders, turbo decoder and SCMA demodulator). The LDPC BP decoders, the polar SC decoders and the turbo decoder are compatible with the inter-frame SIMD strategy while the polar SC/SCL decoders and the SCMA demodulator are compatible with the intra-frame SIMD strategy. Depending on the code families, we focus on different constraints. The LDPC BP decoders have been implemented to support many variants and thus to maximize the flexibility at the cost of lower throughputs and higher latencies compared to other works. This choice enables to evaluate the decoding performance of many algorithmic combinations. In the polar decoders, flexibility as well as aggressive optimizations are considered, combined and compared. The turbo decoder focuses on achieving the highest possible throughputs and some specializations are made for the LTE standard. Finally the SCMA demodulator implementation tries to propose a compromise between high throughputs and low latencies. Most of the proposed software implementations have been published in scientific conferences and journals [2, 3, 4, 5, 6].

The optimizations performed in the proposed implementations are compatible with a large set of CPUs, compilers, and data types. This portability is one of our main concern and we believe that the proposed software implementations will be easily extended to future CPUs as long as there is no drastic changes in the hardware architectures.

Some optimization strategies have not been considered in the proposed implementations and are good candidates to improve them. For each implementation the inter- or the intra-frame SIMD strategy has been selected. With the growing size of the SIMD registers, it becomes difficult

to achieve high efficiency using the intra-frame SIMD strategy. However, the reduced latencies are still very interesting, especially for the SDR and the C-RAN needs. Then, it could be a good idea to combine both the intra- and inter-frame SIMD strategies. The intra-SIMD will absorb as much as possible the algorithm inherent parallelism while the inter-SIMD will fill the empty elements in the SIMD registers. The inter-SIMD also has its limits, when large frames are processed in parallel, the amount of required memory linearly increases with the frame size. This leads to reduced throughput efficiency as it will be shown in Chapter 4. Additionally, for some decoding algorithms, it is not possible to maintain optimal error-rate performance with too short fixed-point representations (8-bit). It could be interesting to consider mixed-precision in these specific cases.

The next chapter introduces AFF3CT, the toolbox we designed to integrate all the proposed implementations in a consistent, modular and extensible forward error correction framework.

3 AFF3CT: A Fast Forward Error Correction Toolbox

This chapter is dedicated to the introduction of our AFF3CT open-source toolbox. The first section describes the main prerequisites driven by four objectives: high performance software implementation, support for algorithmic heterogeneity, portability and reproducibility. In the second section, AFF3CT is compared with the other existing `C`/`C++` FEC software libraries. The third section presents AFF3CT as a library dedicated to the digital communication algorithms. The software architecture and functionalities are described. Then, examples of library use are given in `C++` and MATLAB®. The fourth section focuses on the AFF3CT BER/FER simulator that comes with the toolbox. A tour of the possible explorations is given and our BER/FER comparator is presented. At the end, the AFF3CT testing strategy is explained. The fifth section shows the impact of AFF3CT in industrial and academic contexts. A review of the scientific publications that used AFF3CT is given. The last section concludes this chapter.

3.1 Prerequisites

AFF3CT for *A Fast Forward Error Correction Toolbox* is a set of tools regrouping all the contributions of this thesis and more. In this section, we explain how AFF3CT answers to the different problematics defined in Section 1.5.

3.1.1 High Performance Implementations

The signal processing community mainly writes source codes with high level languages like MATLAB® or Python. These languages enable to write implementations close to the pseudo-code but an important part of the CPU computational power is wasted. When targeting low error-rate functional simulations or real-time constraints (like for the C-RAN and the SDR), these high level languages are not suitable. With the increasing complexity of the digital communication systems it becomes crucial to have high performance implementations.

To this purpose, AFF3CT is mainly written in `C++` [Str13]. This choice has been made to focus on high performance implementations without sacrificing too much the expressiveness. `C++` is a compiled language, it enables very low level programming paradigms like intrinsics functions (or even assembly code) as well as high level concepts like the Object-Oriented Programming (OOP) paradigm. Moreover, `C++` comes with the template meta-programming technique to facilitate the programming at compile time. Another main advantage of `C++` is that it is constantly evolving [Str20] and it is well-spread in the HPC community. However, we choose to limit the utilization of `C++` to its 2011 version (`C++11`). This choice has been made for two main reasons: 1) to maximize the compatibility with the installed compilers in various environments; 2) `C++11` features are sufficient for digital communication systems.

For the signal processing algorithms implemented in AFF3CT, we observed that compared to interpreted languages, the speedups range from 10 to 1000 in `C++`. If we consider the heavily optimized implementations presented in Chapter 2, the speedups are closer to 1000. Of course the speedups are not simply coming from the porting of the MATLAB® code to the `C++` code. It is true that the compiler is sometime able to perform optimizations that can benefit for the overall performance. But, most of the speedup comes from dedicated implementations. `C++` enables source code implementations to take advantage of the hardware architecture. Moreover, it is up to the developer to exploit his/her knowledge of the CPU architecture in the design process. As explained in Chapter 2, optimizations like the vectorization, the choice of an adapted data layout and the loop unrolling are the keys of the proposed high performance software implementations.

3.1.2 Support for Algorithmic Heterogeneity

As shown in Chapter 1 and 2, there are many signal processing algorithms along with many possible implementations. A summary list of the algorithms supported by AFF3CT is given in the next section. It motivates the need to regroup and package all these algorithms in a common toolbox. The main interests are 1) to propose common and homogeneous interfaces to the users and 2) to maximize code reuse among implementations.

In the context of the channel codes, the algorithmic heterogeneity is challenging. Actually, each family has its own specificities. This is why in most existing projects the focus is made on a single code family (see Section 3.2). This strongly motivated the need of a toolbox like AFF3CT. The objective is to homogenize the use of various FEC code families.

MIPP is an example of code reuse as it defines elementary blocks used everywhere in AFF3CT. Other macro blocks are also often reused like the reordering process proposed in Sections 2.2.2. There are many other macro blocks in AFF3CT similar to the ones that are presented in the manuscript. This enables to speedup and facilitate the implementation of new efficient algorithms.

3.1.3 Portability

Portability is a main concern in AFF3CT. The signal community use multiple operating systems. The predominant ones are Windows, macOS and Linux. Thanks to the `C++11` standard library, the same AFF3CT source code can be compiled on these three operating systems. It is possible to compile with the GNU compiler (GCC), the Clang compiler, the Intel® `C++` compiler (ICPC) and the Microsoft® Visual compiler (MSVC). Note that other operating systems and compilers may also work as long as they are compatible with the `C++11` standard.

AFF3CT also takes advantage of various common CPU architectures like Intel®/AMD® and ARM® processors. The Intel®/AMD® CPUs are widely spread in current laptops as well as in the clusters (or in the supercomputers). ARM® CPUs are interesting as they are generally consuming less energy than Intel®/AMD® CPUs. They are good candidates for embedded systems. Moreover, they are more and more present in HPC contexts. At the time of the writing, Fugaku, the most powerful supercomputer in the world, is based on ARM CPUs.

The heterogeneity of the CPU architectures is mainly managed by MIPP (see Section 2.1). The compiled binary is dedicated to the appropriate SIMD ISA. If the architecture is not recognized then the AFF3CT binary falls back to a sequential version.

3.1.4 Reproducibility

In the signal processing community it is not common to share the resulting implementations of a scientific publication. Thus, it is sometime a tedious task to reproduce the state-of-the-art results. Consequently, the community spends a non-negligible amount of time in "reinventing the wheel". We think this should be avoided, thus AFF3CT is an open source toolbox coming with a permissive MIT license. This way, industrial and academic actors can invest themselves and reuse parts of AFF3CT in their own projects without any restrictions. The diffusion of AFF3CT is discussed in Section 3.5.

Even when a code is fully open, there is no guarantee that the achieved results can be reproduced as the code is constantly evolving. Any modification of the source code can break features that were working before. This problem is inherent to all living projects. To prevent regressions as much as possible, a full pipeline of tests has been created. It is detailed in Section 3.4.4. Each time someone makes a modification on the AFF3CT source code, then the pipeline of tests is triggered. The reproducibility of the results is based on the fact that for a given AFF3CT simulator command line, the output BER/FER decoding performance should always be the same. In other terms, the AFF3CT simulator is deterministic.

3.2 Related Works

In the digital signal processing community, many researchers implement their own simulation chain to validate their works. Table 3.1 presents, to the best of our knowledge, a list of currently

Table 3.1 – C/C++ open source channel coding simulators/libraries.

Name	Ref.	Contri-butors	Code Lines	Start Year	License	Polar	LDPC	Turbo	Turbo P.	BCH	RS	Conv.	RA	Rep.	Erasure
AFF3CT	[1]	11	76k	2016	MIT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
aicodix GmbH	[aic18]	1	7k	2018	Copyright	✗	✓	✗	✗	✓	✓	✗	✗	✗	✗
eccpage	[Mor89]	20	-	1989	-	✗	✓	✓	✗	✓	✓	✓	✗	✗	✓
EZPWD	[Kun14]	2	6k	2014	GPLv3	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
FastECC	[Zig15]	2	1k	2015	Apache 2.0	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
FEC-AL	[Tay18a]	1	3k	2018	BSD	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓
FECpp	[Llo09]	1	2k	2009	-	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
GNURadio	[Ron+06]	192	270k	2006	GPLv3	✓	✓	✗	✗	✗	✗	✓	✗	✓	✗
Inan	[IS18]	2	13k	2018	Copyright	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
IT++	[COP+05]	20	109k	2005	GPLv3	✗	✓	✓	✗	✓	✓	✓	✗	✗	✗
Le Gal	[Le 15]	1	83k	2015	-	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
Leopard	[Tay+17]	4	5k	2017	BSD	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
libcorrect	[Arm+16]	6	5k	2016	BSD	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗
Neal	[Nea06]	1	5k	2006	Copyright	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
OpenAir	[EUR13]	148	740k	2013	OAI Public	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
OpenFEC	[Cun+09]	8	55k	2009	CeCCIL-C	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗
Schifra	[Par10]	1	7k	2010	GPLv3	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
Siamese	[Tay18b]	1	11k	2018	BSD	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓
Tavildar (Polar)	[Tav16b]	1	2k	2016	-	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Tavildar (LDPC)	[Tav16a]	1	1k	2016	-	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
the-art-of-ecc	[Mor06]	1	-	2006	Copyright	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗
TurboFEC	[PT+15]	2	4k	2015	GPLv3	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗

available C/C++ open source channel coding simulators/libraries. This comparison table is also available online where it is regularly updated [1]. We choose to compare with projects compiled as binaries, since they aim at high throughput and low latency, as AFF3CT. Many open source projects in Python or in MATLAB® exist as well. But these tools are usually slower than compiled binaries, and rather aim at prototyping.

Table 3.1 shows that, generally, the C/C++ FEC libraries target a single family or a small subset of channel codes. As a consequence, a large effort is spent to re-develop similar features, since all those libraries and tools share many characteristics (except the channel code itself). AFF3CT attempts to lower this redundancy by offering a full simulator/library that consistently supports a wide range of channel codes and homogenizes usage (command line, C++ interfaces, etc.) across all code families. One can observe that AFF3CT is the only library to support the LDPC codes, the polar codes and the turbo codes. These three channel codes are known to be the most challenging ones to implement.

3.3 Library of Digital Communication Algorithms

3.3.1 Software Architecture

AFF3CT is developed in C++ in an object-oriented programming style. It provides fundamental classes involved in the building of digital communication chains. For instance, in Fig 1.14, the source, the encoder, the modulator, the channel, the demodulator, the decoder and the monitor are module classes. Tools can be classes or functions. For instance, the polar API presented in Section 2.5.2 is a static class implementing the polar `f`, `g` and `h` functions (see Equation 1.7) that are common to all the polar decoders. Many implementations of defined interfaces can coexist. For instance, the abstract `Encoder` class that defines the `encode` pure virtual method. The

1. C/C++ Open Source FEC Libraries: https://aff3ct.github.io/fec_libraries.html

`encode` method takes a vector $\mathbf{u}$ of K bits as inputs and outputs a vector $\mathbf{c}$ of N bits. Then, there are many implementations of the `Encoder` class like the `Encoder_polar` class, the `Encoder_LDPC` class, the `Encoder_turbo` class, etc. To simplify the instanciation of the non-static classes (like the encoders), many factory classes have been created. Their job is to simplify the allocation of the module and tool objects.

3.3.1.1 Module

All the classes that implement communication chain elements inherit from the `Module` abstract class. The particularity of the module classes is that they have to expose at least one method that can be called in the context of digital communication chains. These specific methods are called *tasks*. A task is an elementary processing performed on some data. For instance, the `Encoder` class inherits from the `Module` class. It also defines the `encode` method which is a task. To be recognized as a task, the `encode` method has to be registered in the constructor of the `Encoder` class. The abstract `Module` class defines and implements a set of functions to perform this registering. Most of the time, when a developer wants to add a new module, he does not need to register any task because it is already done for him. For instance, if a developer wants to add a new encoder, then he simply needs to inherit from the proposed `Encoder` class and to implement the `encode` method. A task is characterized by its *sockets*. They are used to describe the input and output data of the task following a philosophy close to *ports* in component-based development approaches. The socket type can be input or output. The sockets also enable to automatically allocate the data. In AFF3CT the convention is to automatically allocate the data of the output sockets.

3.3.1.2 Tools

The tools regroups many different types of processing:

— **Algorithmic & Math**: the algorithmic components focus on the implementation of traditional algorithmic structures. For instance, this contains implementations of trees, matrices, histograms, etc. It also contains sorting and PRNG algorithm implementations. The math components regroup the interpolations, the Galois fields, the distributions, the integrations, etc.;

— **Channel code**: this type of tools regroups the processing implementations that are common to a single code family. For instance, the `Polar_API` is located here;

— **Display**: these components are classes and functions dedicated to the display of the information in the terminal or in files. Statistic functions are located here as well as classes dedicated to the display of the BER and the FER performances;

— **Interface**: these abstract classes define interfaces. For instance, the `Interface_reset` proposes a common interface for the `reset` method. This way, all the classes inhering from this interface have exactly the same prototype for their implementation of the `reset` method;

— **Performance**: this type of components is dedicated to high performance implementation. The vectorized reordering process presented in Section 2.2.2 is located here.

The above list is not exhaustive but is intended to give a representative overview of what can be found in the different tools.

3.3.1.3 Factory

In AFF3CT, one has to allocate modules dynamically at runtime. Combined with the fact that there are many possible combinations of modules, we applied the factory pattern. In the OOP paradigm, the factory method pattern is dedicated to the the problem of creating objects without having to specify the exact class of the object that will be created. This is done by calling a method on the factory. In AFF3CT, all the factories propose a `build` method to this purpose. The return type of the `build` method is always an abstract class that regroups a sub-set of implemented classes. Considering the factory `Source` class, first the instantiation of this class is required. Then, the created object comes with a list of public members that can be manually set or deduced from the command line arguments. Once this is done the `build` method can be called. This method makes use of the public members (previously set) of the object to instantiate an object of the module `Source` class. This object can be a `Source_random` object, a `Source_user_binary` object, etc.

3.3.2 Examples of Library Use

As a FEC library, AFF3CT can be used programmatically in real-time contexts or to build specific functional simulations. AFF3CT blocks can also be operated in external projects without restriction. In this section we propose two illustrative examples of the AFF3CT library usage. The first one is dedicated to the simulation of a digital communication chain while the second focuses on the validation of a hardware decoder.

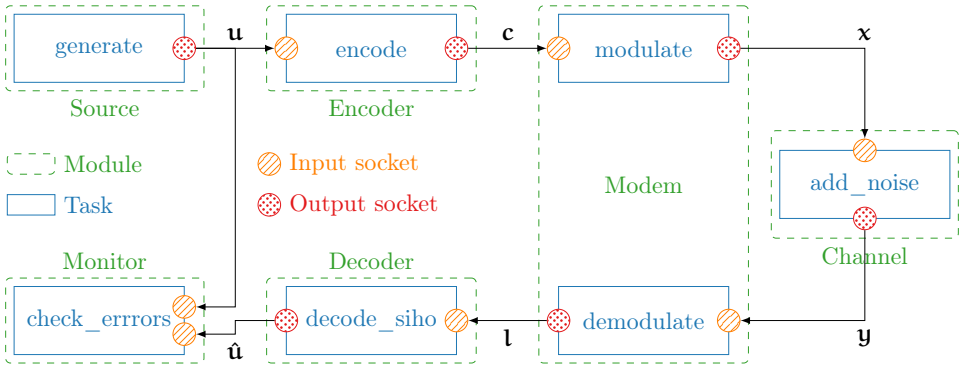


Figure 3.1 – Simulation of a digital communication chain using the AFF3CT library.

Simulation of a Digital Communication Chain A simulation chain that can be implemented is given in Figure 3.1. The represented modules an tasks correspond to the classes and methods presented in the previous section. For instance, the *modem* module contains the *modulate* and *demodulate* tasks. Figure 3.1 presents common modules and tasks typically found in a typical digital communication chain. It shows that the number of tasks per module can vary depending on the module type. The first step is to allocate the modules. In Listing 3.1 we chose to allocate modules on the stack. But it is also possible to do the same on the heap. K is the number of information bits, N is the frame size and E is the number of erroneous frames to simulate. In this basic example, a repetition code is selected, it simply repeats the information bits N/K times. The next step is to bind the sockets of successive tasks together (see Listing 3.2). To propose an easy to use interface, sockets and tasks can be selected through the `[]` operator, which takes a `C++` strongly typed enumerate. This way it is possible to specialize the code depending on whether it is a socket or a task. Strongly typed enumerates are checked at compile time (contrary

```
#include <aff3ct.hpp>
using namespace aff3ct;

constexpr int K = 128; // number of information bits
constexpr int N = 256; // codeword or frame size
constexpr int E = 100; // number of errors to simulate

// allocate the module objects
module::Source_random<>      Src(K   );
module::Encoder_repetition<> Enc(K, N);
module::Modem_BPSK<>         Mdm(N   );
module::Channel_AWGN<>       Chn(N   );
module::Decoder_repetiton<>  Dec(K, N);
module::Monitor_BFER<>       Mnt(K, E);
```

Listing 3.1 – Example of modules allocation with the AFF3CT library.

```
// bind the sockets over the tasks
Enc[module::enc::sck::encode      ::u ].bind(Src[module::src::sck::generate   ::u]);
Mdm[module::mdm::sck::modulate    ::c ].bind(Enc[module::enc::sck::encode     ::c]);
Chn[module::chn::sck::add_noise   ::x ].bind(Mdm[module::mdm::sck::modulate   ::x]);
Mdm[module::mdm::sck::demodulate  ::y ].bind(Chn[module::chn::sck::add_noise  ::y]);
Dec[module::dec::sck::decode_siho ::l ].bind(Mdm[module::mdm::sck::demodulate ::l]);
Mnt[module::mnt::sck::check_errors::u1].bind(Src[module::src::sck::generate   ::u]);
Mnt[module::mnt::sck::check_errors::u2].bind(Dec[module::dec::sck::decode_siho::u]);
```

Listing 3.2 – Example of sockets binding with the AFF3CT library.

```
// the simulation loop
while (!monitor.fe_limit_achieved()) {
    Src[module::src::tsk::generate    ].exec();
    Enc[module::enc::tsk::encode      ].exec();
    Mdm[module::mdm::tsk::modulate    ].exec();
    Chn[module::chn::tsk::add_noise   ].exec();
    Mdm[module::mdm::tsk::demodulate  ].exec();
    Dec[module::dec::tsk::decode_siho ].exec();
    Mnt[module::mnt::tsk::check_errors].exec();
}
```

Listing 3.3 – Example of tasks execution with the AFF3CT library.

to standard enumerates), making it impossible to use wrong values. For instance, in the example, the *source* module output socket `module::src::sck::generate::u` is connected to the input socket `module::enc::sck::encode::u` of the *encoder*. The simulation is then started and each task is executed. In Listing 3.3, the whole communication chain is executed multiple times, until the $\mathtt{E} = 100$ frame error limit is reached. Complete examples are available on GitHub [2].

Validation of a Hardware Decoder The AFF3CT library has also been used to prototype FPGA decoders. In [8], a BCH decoder is implemented on a Xilinx® Artix-7 FPGA. AFF3CT simulates the transmission of a first frame. The noisy frame is then sent to the FPGA using the UART protocol. The hardware BCH decoder processes the frame and sends it back to the PC. AFF3CT can then perform the rest of the processing for this frame. Once the number of residual errors is updated, AFF3CT starts with a new frame, and so on. The decoding performance matches to the performance of the pure software simulation which shows that the hardware BCH decoder is correctly implemented. This process is also known as *hardware in the loop*.

3.3.3 MATLAB® Wrapper

In the signal processing community it is common to exploit MATLAB® to implement and to evaluate new algorithms or/and configurations. More specifically, the *Communications Toolbox* is often used. This toolbox contains a larger set of digital communication algorithms than AFF3CT. But they often lack of efficiency, especially for the channel decoder implementations. Knowing that, a MATLAB® wrapper has been proposed to be interfaced to the compiled AFF3CT library. The wrapper can be seen as a new MATLAB® toolbox that proposes faster decoder implementations than the traditional MATLAB® communications toolbox. AFF3CT also comes with channel decoders that are not available in the standard communications toolbox.

```
K = 128; N = 256; E = 100;

Src = aff3ct_module_source_random     (K   );
Enc = aff3ct_module_encoder_repetition(K, N);
Mdm = aff3ct_module_modem_BPSK        (N   );
Chn = aff3ct_module_channel_AWGN      (N   );
Dec = aff3ct_module_decoder_repetition(K, N);
Mnt = aff3ct_module_monitor_BFER      (K, E);

while ~Mnt.fe_limit_achieved()
    u = Src.generate    (    );
    c = Enc.encode      (u   );
    x = Mdm.modulate    (c   );
    y = Chn.add_noise   (x   );
    l = Mdm.demodulate  (y   );
    v = Dec.decode_siho (l   );
        Mnt.check_errors(u, v);
end
```

Listing 3.4 – Example of the AFF3CT MATLAB® wrapper.

The proposed MATLAB® wrapper is automatically generated for the AFF3CT headers. The Clang compiler is used to generate the Abstract Syntax Tree (AST) of the AFF3CT source

2. AFF3CT library examples: https://github.com/aff3ct/my_project_with_aff3ct/

code. A Python script extracts useful classes and methods. These data are stored in a JSON database. Then, another Python script has been written to generate `C++` and MATLAB® codes. It effectively performs the interfaces between the AFF3CT library and MATLAB®. Listing 3.4 shows the same example of code as in Section 3.3.2 written in MATLAB®. At the time of the writing, the MATLAB® wrapper has not been publicly released yet.

3.3.4 Software Functionalities

The AFF3CT software functionalities are decomposed in three main parts: the *codecs*, the *modems* and the *channels*.

Table 3.2 – List of the channel codes (codecs) supported in AFF3CT.

Channel Code	Standard	Decoders
LDPC	5G (data), Wi-Fi, WiMAX, WRAN, 10 Gigabit Eth., DVB-S2, CCSDS etc.	Scheduling: Flooding and H./V. Layered Sum-Product Algorithm (SPA, log-SPA) Min-Sum its derivatives (MS, NMS and OMS) Approximate Min-Star (AMS) Bit Flipping: GallagerA/B/E, PPBF, WBF
Polar (Arıkan mono-kernel)	5G (control channel)	Successive Cancellation (SC) Successive Cancellation List (SCL) CRC-Aided SCL (CA-SCL, FA-SCL, PA-SCL) Soft Cancellation (SCAN)
Polar (mono/multi-kernel generic)	–	Successive Cancellation (SC) Successive Cancellation List (SCL) CRC-Aided SCL (CA-SCL, PA-SCL)
Turbo (single and double binary)	LTE (3G, 4G), DVB-RCS, CCSDS, etc.	Turbo BCJR Turbo BCJR + Early Termination (CRC) Post proc.: Flip aNd Check (FNC)
Product	WiMAX (opt.)	Turbo Chase-Pyndiah
BCH	CD, DVD, SSD, DVB-S2, Bitcoin, etc.	Berlekamp-Massey + Chien search
Reed-Solomon	CD, DVD, SSD, DVB-T, ADSL, etc.	Berlekamp-Massey + Chien search
Convolutional (single and double binary)	NASA	BCJR - Maximum A Posteriori (MAP) BCJR - Linear Approximation (L-MAP) BCJR - Max-log Approximation (ML-MAP)

The codecs are the main part of the toolbox. There is a broad range of supported codes listed in Table 3.2. They naturally encompass the encoders and decoders. But they can also include puncturing patterns to shorten frame length according to some communication standards. Most of the codec algorithms come from the literature, while the others have been designed under AFF3CT [Ton+16b, Ton+16a, Ton17, 3]. In channel coding, the decoder is the most time-consuming process, compared to the puncturing and the encoding processes. This is why a specific effort is put on ensuring the high computing performance of the decoders. Most of the decoding algorithms have thus been optimized to satisfy high throughput and low latency

constraints [LLJ15, 4, 6, 5]. Those optimizations generally involve a vectorized implementation, a tailored data quantization and the use of fixed-point arithmetic.

Table 3.3 – List of the modulations/demodulations (modems) supported in AFF3CT.

Modem	Standard	Information
N-PSK	IEEE 802.16 (WiMAX) UMTS (2G, 2G+) EDGE (8-PSK), ...	Phase-Shift Keying
N-QAM	IEEE 802.16 (WiMAX) UMTS (2G, 2G+) 3G, 4G, 5G, ...	Quadrature Amplitude Modulation
N-PAM	IEEE 802.16 (WiMAX) UMTS (2G, 2G+) 3G, 4G, 5G, ...	Pulse Amplitude Modulation
CPM	GMSK, Bluetooth IEEE 802.11 FHSS	Continuous Phase Modulation Coded (convolutional-based) modulation
OOK	IrDA (Infrared) ISM bands	On-Off Keying Used in optical communication systems
SCMA	Considered for 5G	Sparse Code Multiple Access Multi-user modulation
User defined	-	Constellation and order can be defined from an external file

In typical communication chains, it is necessary to adapt the digital signal to the physical support. This operation is performed by the modulator and conversely by the demodulator. AFF3CT comes with a rich set of modems to this purpose. Table 3.3 lists all the supported modems. AFF3CT supports several coded modulation/demodulation schemes like the Continuous Phase Modulation (CPM) [AS81, ARS81] and the Sparse Code Multiple Access (SCMA) modulation [NB13, Gha+17, 2] with many codebooks [Alt15b, WZC15, CWC15, Zha+16, KS16, SWC17, KS17]). It enables to easily combine and evaluate the channel codes with several types of modulations. In the case of the CPM, analogical wave shapes are also simulated. The other modulation schemes are at the digital level.

Table 3.4 – List of the channel models supported in AFF3CT.

Channel	Multi-user	Information
AWGN	Yes	Additive White Gaussian Noise
BEC	No	Binary Erasure Channel
BSC	No	Binary Symmetric Channel
Rayleigh	Yes	Flat Rayleigh fading channel
User defined	No	User can import noise samples from an external file

For simulation purposes, it is crucial to emulate the behavior of the physical layer. This is the role of the channel. There are many possible configurations depending on the physics phenomena

to simulate. Table 3.4 reports all the supported channels. The channels involve complex floating-point computations. It is frequent to use expensive exponential and trigonometric operations. As for the decoders, the channel software implementations have to be carefully optimized based on branch instructions reduction and massive vectorization. The *multi-user* column refers to the ability of the channel to add correlated noise to a sub-set of frames.

3.4 Simulation of Digital Communication Algorithms

3.4.1 A Simulator Application on Top of the Library

The AFF3CT toolbox comes with a dedicated functional simulator [9]. It is based on the AFF3CT library presented before. We remarked that the functional simulation chains are often similar. For this reason, the simulation chain presented in Figure 3.1 has been implemented and enriched. The proposed simulator supports multi-threading to take advantage of current CPU multi-core architectures. It is also able to run on supercomputers and comes with a multi-node implementation based on the well-known HPC Message Passing Interface (MPI). The multi-core and multi-node performance of the AFF3CT simulator is illustrated later in Section 4.6.

One of the main advantage of the AFF3CT simulator is to come with a common interface for many channel code families. It is also possible to evaluate the error-rate performance of these code families on various configurations thanks to the supported modems and channel models. This makes it easy to compare different code families with each other.

```
$ aff3ct -C "POLAR" -K 1723 -N 2048 -m 1 -M 4 -s 1 --dec-type "SC"
```

Listing 3.5 – Example of an AFF3CT simulator command.

The AFF3CT simulator is a command line executable. All its possible parameters are exhaustively documented [3]. Listing 3.5 proposes to simulate a $(2048, 1723)$ polar code from 1 dB to 4 dB with a step of 1 dB (see Section 1.4.1). By default, the AWGN channel is selected as well as the BPSK modulation. Then the SC decoder is specified. For a given SNR, by default the simulation stops when more than 100 erroneous frames have been detected.

```
1  # ----------------------------------------------------
2  # ---- A FAST FORWARD ERROR CORRECTION TOOLBOX >> ----
3  # ----------------------------------------------------
4  # Parameters :
5  # [...]
6  # ----------|----------||----------|----------|----------|----------|----------
7  #     Es/N0 |    Eb/N0 ||      FRA |       BE |       FE |      BER |      FER
8  #      (dB) |     (dB) ||          |          |          |          |
9  # ----------|----------||----------|----------|----------|----------|----------
10       0.25 |     1.00 ||      104 |    16425 |      104 | 9.17e-02 | 1.00e+00
11       1.25 |     2.00 ||      104 |    12285 |      104 | 6.86e-02 | 1.00e+00
12       2.25 |     3.00 ||      147 |     5600 |      102 | 2.21e-02 | 6.94e-01
13       3.25 |     4.00 ||     5055 |     2769 |      100 | 3.18e-04 | 1.98e-02
14 # End of the simulation.
```

Listing 3.6 – Example of an AFF3CT simulator output.

3. AFF3CT documentation: https://aff3ct.readthedocs.io

Listing 3.6 shows the simulation results corresponding to the AFF3CT command given in Listing 3.5 (note that some details have been removed for concision). The same command line always gives the same decoding performance results. `FRA` stands for the number of simulated frames, while `BE` and `FE` are the number of bit and frame errors. The simulator output is adapted to post processing: lines starting with a hashtag can be skipped.

3.4.2 In-depth Parameter Exploration

One of the main strength of the AFF3CT simulator is to enable the exploration of various configurations. In this section, a tour of possible experimentation scenarios is given. The objective is not to be exhaustive and many more parameters could be explored. However, it gives a representative overview of the large variety of parameters that can be tweaked in the proposed simulator. As shown in Section 3.3.4, many code families are supported. To the best of our knowledge the AFF3CT toolbox regroups more channel codes than all the other existing libraries (see Section 3.2). Each of these codes can be simulated over many channel models (BEC, BSC, AWGN and Rayleigh) and modulation schemes (PSK, QAM, PAM, OOK, CPM and SCMA). In this thesis the channel model is always the AWGN and the modulation scheme is almost always a BPSK. It can also be the SCMA modulation. It will be explicitly mentioned in the latter case. For each channel code, many decoding algorithms and their corresponding approximations can be compared.

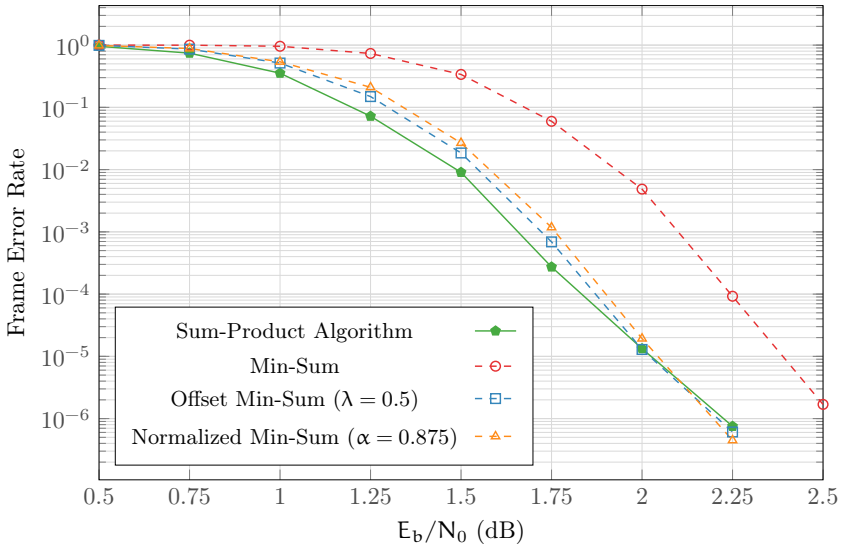


Figure 3.2 – Decoding performance of the LDPC BP algorithm depending on the update rules (horizontal layered scheduling). 40 iterations, IEEE 802.16e (WiMAX) $\mathcal{H}$ parity matrix ($N = 2304$, $R = 1/2$).

Impact of the Decoder Algorithmic Parameters on the FER In Figure 3.2, the LDPC belief propagation (BP) decoding algorithm is considered with an horizontal layered scheduling. The $\mathcal{H}$ parity matrix has been taken from the WiMAX standard ($N = 2304$, $R = 1/2$). The impact of various update rules on the decoding performance is observed. As explained in Section 1.3.2, the Min-Sum (MS) is an approximation of the Sum-Product Algorithm (SPA) and leads to a performance loss. The Offset Min-Sum (OMS) and the Normalized Min-Sum (NMS) are improvements of the MS. They enable to recover a part of the SPA decoding performance.

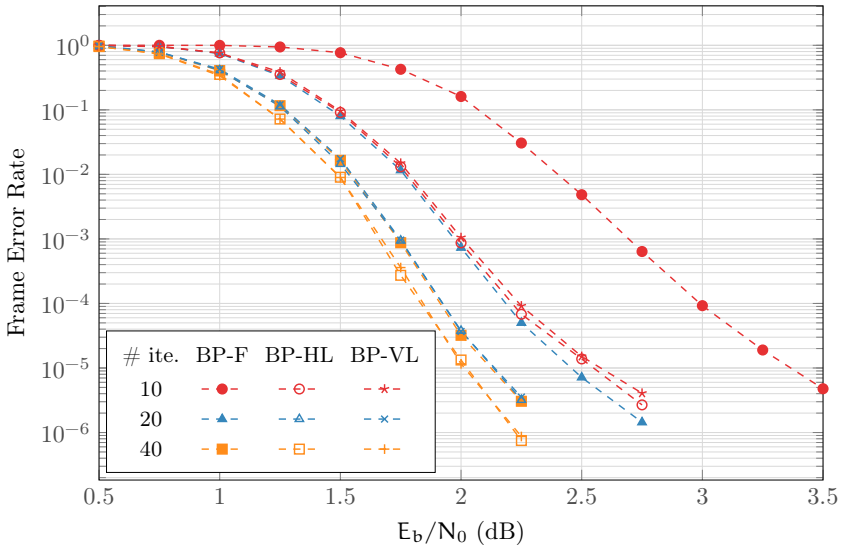


Figure 3.3 – Decoding performance of the LDPC BP algorithm depending on the scheduling techniques. Flooding (BP-F), horizontal layered (BP-HL) and vertical layered (BP-VL) scheduling are considered with SPA update rules. IEEE 802.16e (WiMAX) $\mathcal{H}$ parity matrix ($N = 2304$, $R = 1/2$).

In Fig 3.3 only the SPA is considered and the decoding performances of various scheduling policies are compared. The results show that the convergence of the layered scheduling policies is faster than the traditional flooding scheduling for a same number of iterations. Increasing the number of iterations improves the decoding performance while it increases the computation complexity of the decoder. It is up to the system designer to chose the right configuration.

Impact of the Decoder Type on the Throughput An example of polar decoders working on a $N = 2048$ and $K = 1723$ code is given in Figure 3.4. The FA-SSCL and PA-SSCL decoders have the same decoding performance for a list size L. However, the throughputs are different depending on the SNR values. This is another example of possible explorations with the AFF3CT simulator. Depending on the targeted SNR range, it is more interesting to choose either decoder.

Impact of the Decoder Quantization on the FER Another important feature is the impact of the quantization on the decoding performance. To increase the throughput or to decrease the latency of a signal processing, it is common to reduce the amplitude of the data. A fixed-point representation can be shorter and more efficient than a floating-point representation. This is true for hardware decoder implementations as well as for high performance software implementations. In Figure 3.5, the longest turbo code from the LTE standard is proposed ($K = 6144$, $R = 1/3$). The same code is evaluated over 3 different data representations. *float* is a 32-bit decoder working on floating-point data, this is the reference. *int-16* and *int-8* decoders are working on 16 bits and on 8 bits, respectively. The $Q_{s,\nu}$ corresponds to the quantization format of the decoder input LLRs (see Equation 2.1). s is the number of bits of the quantized number, including ν bits for the fractional part. The quantization format is a parameter of the AFF3CT simulator. In Figure 3.5, the 16-bit quantization is able to match the reference decoding performance while there is a little performance degradation in 8-bit.

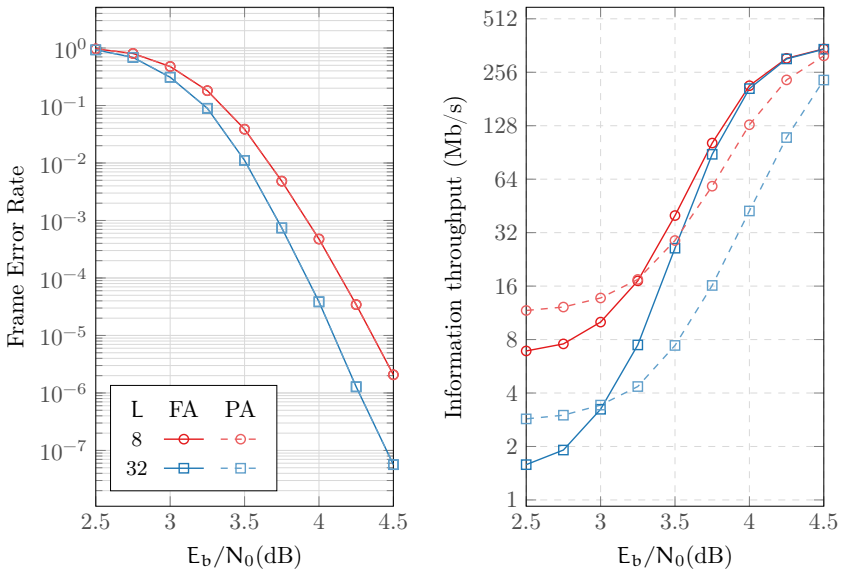


Figure 3.4 – Frame Error Rate performance and throughput of the polar Fully and Partially Adaptive SSCL decoders (FA and PA). $N = 2048$, $K = 1723$ and 32-bit CRC (GZip). Throughputs have been measured on the Intel® Core™ i5-6600K CPU.

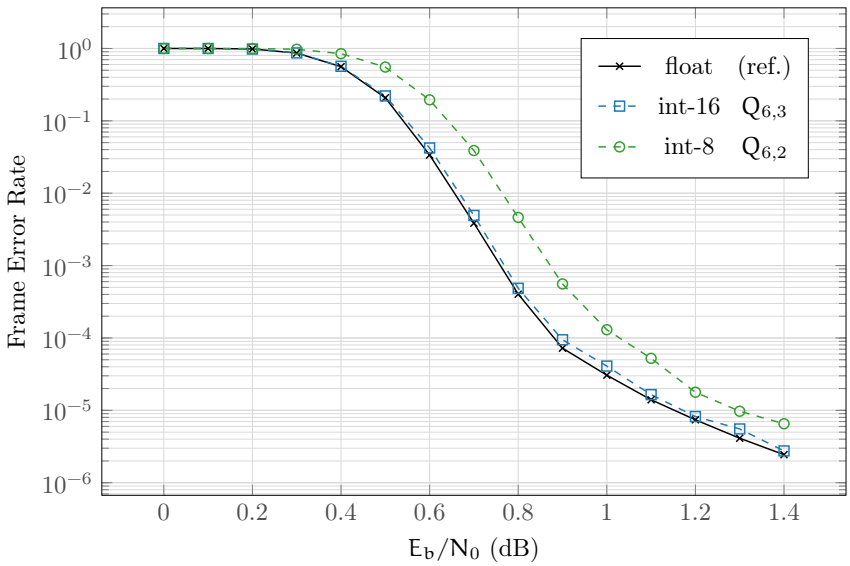


Figure 3.5 – Frame Error Rate of the turbo decoder for $K = 6144$, $R = 1/3$ and 6 decoding iterations. Enhanced max-log-MAP algorithm ($\alpha = 0.75$).

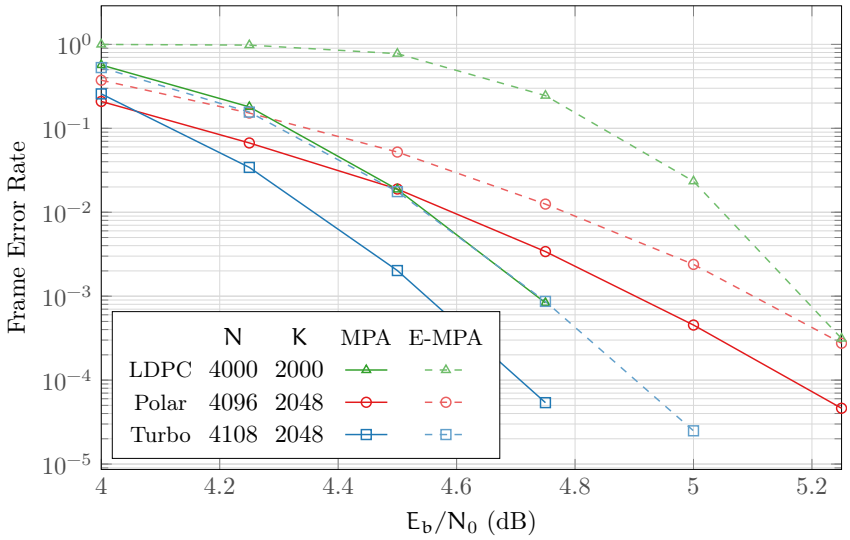


Figure 3.6 – FER evaluation of the SCMA MPA and E-MPA demodulators combined with LDPC codes, polar codes and turbo codes ($K \approx 2048$ and $R \approx 1/2$).

Effect of the SCMA Modulation Scheme on a Sub-set of Channel Codes More complex simulation scenarios where the BPSK modulation is replaced by the SCMA modulation is illustrated in Figure 3.6. The MPA demodulator and its E-MPA approximation are combined with the LDPC codes, the polar codes and the turbo codes. The LDPC $\mathcal{H}$ parity matrix comes from the MacKay personal webpage [4]. The LDPC decoder used is the BP-HL with the SPA update rules (100 iterations). The polar code is built from the Gaussian Approximation technique. The polar decoder is the CA-SSCL decoder with $L = 32$ (the 32-bit GZIP CRC is used). The turbo code comes from the LTE standard and it is punctured to support $R \approx 1/2$. The turbo decoder is based on the EML-MAP with $\alpha = 0.75$ and 6 iterations. The purpose of these curves is not to directly confront the three channel code families even if we tried to select codes that have close enough characteristics. The results show that the E-MPA approximation leads to a performance degradation for each channel code family. But, this negative impact is higher for the selected LDPC code than for other ones.

3.4.3 BER/FER Comparator and Pre-simulated Results

The AFF3CT output (see Listing 3.6) is not adapted to see the error-rate performance at a glance. One can note that it is even more complicated to compare two or more simulation outputs with each other. Traditionally the BER and FER decoding performances are presented in a form of graphical curves (see Figure 1.15). It is then much easier to compare their decoding performance.

To this purpose, we introduced the BER/FER comparator. It is available online on the AFF3CT website [5]. It is capable of reading the AFF3CT simulator outputs and it can also easily adapt to many other formats. The comparator is written in JavaScript. This enables to run the comparator easily on any web browser (no installation is needed). A database of AFF3CT pre-simulated results is available. This database is the same as the error-rate reference results used in the regression tests (see Section 3.4.4). These references are classified according to different

4. MacKay's webpage: http://www.inference.org.uk/mackay/codes/data.html
5. AFF3CT online BER/FER comparator: https://aff3ct.github.io/comparator.html

characteristics: the code type, the modem type, the channel type, the frame size (N) and the code rate (R). At the time of the writing, approximatively 500 BER/FER references are available. For each reference, it is possible to get the corresponding command line in the AFF3CT simulator. This way, it is easy to reproduce the reference results or to modify the command line parameters. The reference curves that have been published are marked with the Digital Object Identifier (DOI) of the corresponding publication. It is then possible to search a specific result from its DOI in the search bar. With the online BER/FER comparator it is easy to share the selected curves with other people thanks to a permalink (= an URL that contains the information of the selected curves). The default proposed database is the AFF3CT database (error-rate references). But it is also possible to access to the database of the Kaiserslautern University [6].

3.4.4 Continuous Integration and Continuous Delivery

AFF3CT's development leverages a streamlined Continuous Integration (CI) process. Each new commit the version control repository (Git) triggers a comprehensive sequence of tests to catch potential regressions. These tests are combined with Continuous Delivery (CD) tasks to deliver new AFF3CT builds automatically.

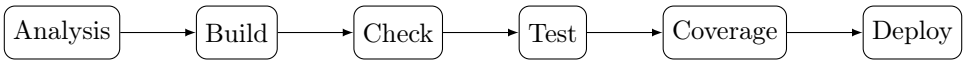


Figure 3.7 – AFF3CT continuous integration and continuous delivery pipeline.

Figure 3.1 shows the different stages of the AFF3CT CI/CD pipeline. The *analysis* stage contains jobs that can be executed without compiling the AFF3CT binaries. The *build* stage is a set of compilation jobs. The *check* stage proposes verification jobs that require the AFF3CT binaries. The *test* stage is composed by a set of jobs focusing on regression testing. Error-rate reference results that have been recorded from previous simulations are replayed. The *coverage* stage measures the percentage of the AFF3CT source code analyzed in the regression tests. Finally, the *deploy* stage contains jobs that are used to push the pipeline results on various targets. For instance, the new builds are automatically pushed on the AFF3CT website [7].

3.5 Impact and Community

AFF3CT is currently used in several industrial contexts for simulation purposes (Turbo concept, Airbus, Thales, Huawei) and for specific developments (CNES, Schlumberger, Airbus, Thales, Orange, Safran), as well as in academic projects (NAND French National Agency project, IdEx CPU, R&T CNES). The MIT license chosen for the project enables industrial and academic partners to reuse parts of AFF3CT in their own projects without any restriction.

AFF3CT has been cited in scientific publications. Many works are exploiting the AFF3CT simulator as a reference for the decoding performance [Pig+18, Pou+18, Gha+18, Wan+19, HV20, RHV20, Duf20]. In other works, AFF3CT has been enriched to support new features. In [Léo+18b] the P-EDGE generator tool (see Section 2.5.3.2) has been modified to generate Transport Triggered Architecture (TTA ≈ VLIW) instructions while in [TB20] a new LDPC code construction method is proposed and directly implemented in the AFF3CT simulator. In some cases AFF3CT is used as a library from which some sub-parts of the toolbox are reused or

6. Kaiserslautern ML BER/FER database: https://www.uni-kl.de/channel-codes/ml-simulation-results/
7. AFF3CT download page: https://aff3ct.github.io/download.html

some methodologies are extracted [Flo18, LCL18, CTG19b, CTG19a, ETG20]. A non-negligible part of the citations are comparisons with the fast decoder implementations described in this manuscript [Zen+17, Léo+18a, GO19, LJ19, She+20]. Finally, in many works, AFF3CT is simply discussed and considered but not directly used [Deb+16b, Deb+16a, Erc+17, NC18, Cen19, KSK19, Vam+19, MMA19, Sha+19, Aly+19, Del+20].

As AFF3CT is open-source, some of the previous works have been integrated inside the toolbox. However, it worths mentioning that AFF3CT is more often simply used than enriched. We believe that its philosophy can lead to a growing community of users and contributors. This is already demonstrated with the increasing activity on the public issue tracker [8].

3.6 Conclusion

In this chapter, AFF3CT, our open-source toolbox dedicated to digital communication algorithms, is presented. First the focus is made on the library with a software architecture that enables the algorithmic heterogeneity. Many channel codes are supported like the LDPC codes, the polar codes, the turbo codes, the TPC codes, the convolutional codes, the BCH codes, the RS codes, etc. To the best of our knowledge, AFF3CT is the library with the most comprehensive support for channel coding algorithms. AFF3CT also comes with multiple channel models (AWGN, Rayleigh, BEC, BSC, etc.) and modulation schemes (PSK, QAM, PAM, OOK, CPM, SCMA, etc.). All these efficient algorithm implementations can be used from interfaces. Examples of library usages are given in native `C++` or by using the MATLAB® wrapper. The AFF3CT toolbox has been valued in a conference [8] and a journal [1].

AFF3CT also comes with a BER/FER simulator. All the previously enumerated features can be simulated over various parameters. The simulator takes advantage of the CPUs multi-core architecture to reduce the restitution time. Its capacity to explore a large variety of parameters is demonstrated. Many parameters can be tweaked like the number of decoding iterations, the approximations in the algorithm implementation, the quantization of the LLRs in the decoders, etc. Some of these parameters are presented according to the decoders and demodulators detailed in Chapter 1 and Chapter 2. Note that this topic has been valued in a national conference [9].

AFF3CT is designed to enable reproducible science. A BER/FER comparator tools has been added to easily search in a database of 500 pre-simulated BER/FER references. All there references are results simulated with AFF3CT and that can be reproduced. To this purpose, a pipeline of tests has been implemented. Each time there is a modification in the source code, the database of references is replayed to avoid regressions. These tests are also ran on multiple architectures (x86 and ARM®) and operating systems (Windows, macOS and Linux) to ensure that the portability is always conserved.

The last section discusses the AFF3CT impact in the community. It is shown that more and more users are adopting the toolbox in both industrial and academic contexts. The application contexts are varied and range from decoding performance validations to the use of specific sub-parts of the library. External contributions are still rare, however.

The next chapter proposes the performance evaluations of the decoder implementations presented in Chapter 2 and packaged in AFF3CT. The overall performance of the BER/FER simulator is also studied on various CPU targets.

8. AFF3CT issue tracker: https://github.com/aff3ct/aff3ct/issues

4 Performance Evaluations and Comparisons

This chapter proposes to evaluate the various contributions exposed in the previous chapters. The three first sections focus on the efficient implementations of the LDPC decoders, polar decoders and turbo decoders. The throughput, the latency and the energy efficiency are studied and compared with other works. The forth section summarizes the most efficient software decoder implementations we found in the literature. Three hall of fames are proposed: one for the LDPC decoders, one for the polar decoders and one for the turbo decoders. Some metrics are defined to facilitate the comparison with the different works. The fifth section is an evaluation of the proposed SCMA demodulator implementations. The throughput, the latency and the energy efficiency are studied over various platforms. The sixth section is a performance analysis of the proposed AFF3CT simulator. A representative digital communication chain is defined and evaluated at two different levels. The first level is the mono-threaded per task performance and the second level is the multi-threaded global performance of the simulator. The last section concludes this chapter.

4.1 LDPC Decoders

In this section we propose to evaluate the fast LDPC BP implementation presented in Section 2.4. The decoder throughputs and latencies are benched on two high-end x86 CPUs and on two $\mathcal{H}$ parity matrices with different characteristics. Then, the proposed BP decoder is compared with the state-of-the-art LDPC decoders.

4.1.1 Experimentation Platforms

Table 4.1 – Specifications of the target processors.

	Platinum 8168	**EPYC 7452**
CPU	Intel® Xeon™ Platinum 8168	AMD® EPYC 7452
Arch.	*Skylake* Q3'17	*Zen 2* Q3'19
Process	14 nm	7 nm
Cores/Freq.	24 cores, 2.7 GHz	32 cores, 2.35 GHz
LLC	33 MB L3	128 MB L3
TDP	205 W	155 W

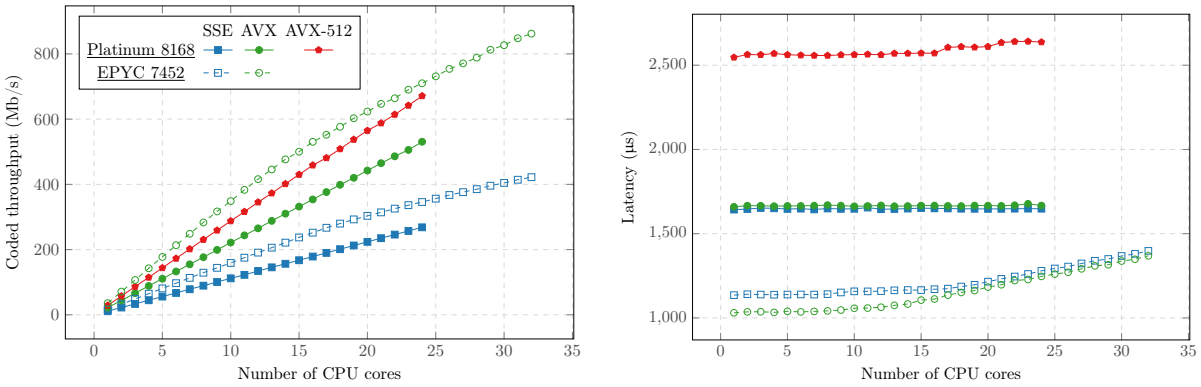
For the experimentations, we selected two high end CPUs: the Intel® Xeon™ Platinum 8168 and the AMD® EPYC 7452 as shown in Table 4.1. The two targets come with a large number of cores, 24 and 32 respectively. The SMT and the frequency boost have been disabled for a matter of reproducibility. The Intel® CPU is able to execute SSE, AVX and AVX-512 instructions while the AMD® CPU can only execute SSE and AVX instructions. The GNU compiler version 7.5 has been used with the following optimization flags: `-O3 -funroll-loops`.

In this section, we choose to evaluate the BP decoder with an horizontal layered scheduling (BP-HL) and with the Normalized Min-Sum (NMS) update rules. We focus on the BP-HL+NMS implementation presented in Section 2.4.2. This decoder is not as flexible as the one presented in Section 2.4.1. But it comes with higher decoding speed. Gains ranging between 20% to 50% are observed depending on the $\mathcal{H}$ parity matrix and on the CPU. In all the presented results, the decoder works on 16-bit fixed-point data. This representation is able to match the BER/FER decoding performance of the floating-point representation. We encounter some difficulties to keep an acceptable level of decoding performance when we ran the 8-bit fixed-point decoder. This is why we have chosen a 16-bit fixed-point representation.

4.1.2 Throughput and Latency Performance on Multi-core CPUs

In this section, we propose to study the throughput and the latency performance evolution depending on the number of cores. Two $\mathcal{H}$ parity matrices are benched. The first one comes from the WiMAX standard and is a middle size matrix where $\mathsf{N} = 2304$ and $\mathsf{R} = 1/2$. The second one is a bigger matrix from the DVB-S2 standard where $\mathsf{N} = 16200$ and $\mathsf{K} = 14400$.

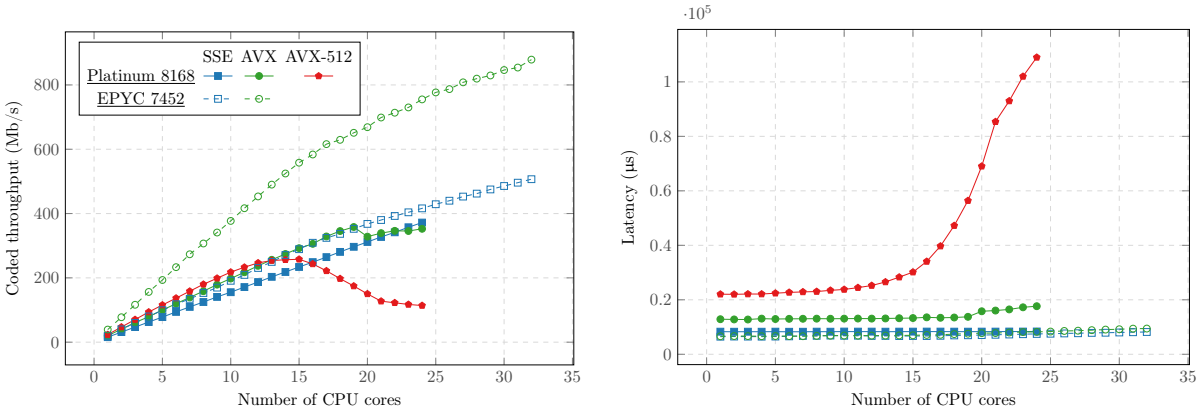
The throughput and latency values for the WiMAX $\mathcal{H}$ parity matrix are given in Figure 4.1. On the Platinum 8168 target, the throughput evolution is almost linear in function of the number of cores and the wider instruction sets. In other words, the best throughput performance is obtained with AVX-512 instructions and on 24 cores. If we look at the latencies, we remark that with AVX-512 instructions there is a marginal increase of the value starting from 17 cores. On the EPYC 7452 CPU the performance increase is mostly linear from 1 to 15 cores. After that, the



(a) Throughput. (b) Latency.

Figure 4.1 – LDPC decoder throughput and latency depending on the number of cores. (2304, 1152) IEEE 802.16e WiMAX code. BP-HL scheduling with 50 iterations and NMS updates rules ($\alpha = 0.875$). 16-bit fixed-point data representation.

latency is increasing and the throughput gains are reduced. Like for the Platinum 8168 target, it is preferable to use the wider possible SIMD instructions on the EPYC 7452 CPU.



(a) Throughput. (b) Latency.

Figure 4.2 – LDPC decoder throughput and latency depending on the number of cores. (16200, 14400) DVB-S2 code. BP-HL scheduling with 50 iterations and NMS updates rules ($\alpha = 0.875$). 16-bit fixed-point data representation.

The obtained throughput and latency values for the DVB-S2 $\mathcal{H}$ parity matrix are presented in Figure 4.2. This matrix is significantly larger than the previous one. As a consequence the memory footprint of the decoder is also higher. This highlights some limitations of the proposed inter-frame SIMD implementation. With this type of implementation, the memory footprint of the decoder is increasing with the size of the SIMD instructions. On the proposed 16-bit implementation, 8 frames are buffered in SSE, 16 frames are buffered in AVX and 32 frames are buffered in AVX-512. On the resulting throughput performance, one can note that the AVX-512 performance is acceptable until a point (14/15 cores) where the performance begin to decrease significantly. At this point the caches of the CPUs are not big enough to contain all the data anymore. The number of slow transactions between the CPU and the RAM are increasing (as well as the number of LLC misses). The same phenomena appears for the AVX implementation on the Platinum 8168 target but later namely 20 cores. On the EPYC 7452 the throughout performance results always take advantage of the increasing number of cores. But in AVX, after

16 cores the performance gains is smaller. This is due to the fact that the data cannot be fully contained in the L2 caches anymore.

The Platinum 8168 comes with higher computational power per core than the EPYC 7452 CPU thanks to its AVX-512 SIMD engine. However the dedicated amount of L3 memory per core is lower than on the EPYC CPU. The Platinum 8168 target dedicates $33/24 = 1.375$ MB per core while the EPYC 7452 target dedicates $128/32 = 4$ MB per core. This is approximately three times more L3 memory per core for the EPYC CPU. As a consequence, the AMD® Zen 2 architecture is more adapted to the inter-frame SIMD strategy.

4.1.3 Comparison with State-of-the-art BP Decoders.

Table 4.2 – Comparison of the proposed BP decoder with the state-of-art. Horizontal layered scheduling. Early termination is disabled. $\mathcal{NT}_c = (\mathcal{T}_c \times \mathtt{i})/(\text{Cores} \times 50)$.

Ref.	Standard $(\mathbf{N}, \mathbf{K})$	Platform	Cores	Pre. (bits)	$\mathtt{i}$	Up. Rules	$\mathcal{L}$ (µs)	$\mathcal{T}_c$ (Mb/s)	$\mathcal{NT}_c$ (Mb/s)
[LJ16]	802.16e $(2304, 1152)$	i7-4960HQ	4	8	50	OMS	1359	217	54.25
[LJ17]	802.16e $(2304, 1152)$	i7-5650U	2	8	10	OMS	12	385	38.50
[Xu+19]	5G $(9126, 8448)$	Gold 6154	18	8	10	OMS	31	4892	54.36
This work	802.16e $(2304, 1152)$	Platinum 8168	24	16	50	NMS	2637	671	27.96
This work	802.16e $(2304, 1152)$	EPYC 7452	32	16	50	NMS	1368	862	26.94

Table 4.2 summarizes the fastest software LDPC BP implementations on CPU we found in the literature. *Pre.* is the precision in bits. $\mathtt{i}$ is the number of decoding iterations. *Up. Rules* are the update rules type. $\mathcal{L}$ is the decoder latency. $\mathcal{T}_c$ is the coded throughput. $\mathcal{NT}_c$ is the normalized coded throughput: this metric considers 50 iterations on a single core. It enables to directly compare the throughput of the listed decoders. In [LJ16], the decoder uses the same inter-frame SIMD strategy as our proposed decoder. The latency is comparable with our implementation while the throughput is about two times higher. This is a direct consequence of the 8-bit quantization. All the presented decoders from the literature are using a 8-bit fixed-point representation while our implementation is executed on 16-bit. The 8-bit implementations require specific modifications to ensure the same level of decoding performance. In the proposed implementation, these specific modifications have not been implemented to the benefit of genericity. Indeed, the same decoder description is able to adapt to 32-bit floating-point and 16-bit fixed-point representations. It is also able to run on various targets like x86 and ARM® CPUs.

In [LJ17, Xu+19], an intra-frame SIMD strategy is used. The parallelism comes from the structure of the $\mathcal{H}$ parity matrix (quasi-cyclic). It is then possible to apply the SIMD instructions during the decoding of a single frame. It leads to much lower latencies. We can see that our proposed decoder is not competitive. The limitation of this type of intra-frame implementation is that the performance strongly depends on the parity matrix. In the worst case, if the $\mathcal{H}$ parity matrix is not quasi-cyclic, then the throughput and the latency cannot be improved.

To summarize, the proposed implementation comes with a throughput approaching to the best implementations ($\approx$ two times slower) while the latency is still very high compared to the intra-frame decoders. One of the main advantage of the proposed implementation is its flexibility. Indeed, it can be run on 32-bit floating-point or 16-bit fixed-point. There is also a unique source code description for the SSE, AVX, AVX-512 and NEON instructions. As it has been shown before, this is valuable because depending on the $\mathcal{H}$ parity matrix, the CPU and the number of cores used. The throughput and latency performances can be more interesting on one or the

other of the SIMD engine. However, even if the proposed implementation is always able to take advantage of the SIMD instructions, we saw some limitations when the memory footprint exceeds the CPU caches.

4.2 Polar Decoders

4.2.1 Successive Cancellation Decoders

In this section we propose to evaluate both the polar SC dynamic and generated decoders presented in Section 2.5.3. First, a study on the software decoders energy efficiency is conducted on low power ARM® CPUs. Then, the impact of the compression technique for the generated SC decoders (see Section 2.5.3.2) is studied. Finally, the dynamic and generated SC decoders are compared with the state-of-the-art decoders.

4.2.1.1 Experimentation Platforms

Table 4.3 – Specification of the x86 platforms.

	E3-1225	**i7-2600**	**i7-4850HQ**
CPU	Intel® Xeon™ E3-1225	Intel® Core™ i7-2600	Intel® Core™ i7-4850HQ
Cores/Freq.	4 cores, 3.1-3.4 Ghz	4 cores, 3.4-3.8 GHz	4 cores, 2.3-3.5 GHz
Arch.	*Sandy Bridge*	*Sandy Bridge*	*Crystal Well*
Process	32 nm	32 nm	22 nm
LLC	L3 6 MB	L3 8 MB	L3 6 MB

Table 4.4 – Specification of the ARM® platforms.

	A15-J	**A15-O/A7-O**	**A57/A53**
SoC	Nvidia® Jetson TK1	Hardkernel® ODROID-XU	ARM® Juno
Arch.	32-bit, *ARMv7*	32-bit, *ARMv7*	64-bit, *ARMv8*
Process	28 nm	28 nm	unspecified (32/28 nm)
big	4xCortex-A15 MPCore freq. 2.32 Ghz L2 1 MB	4xCortex-A15 MPCore freq. 0.8–1.6 GHz L2 2 MB	2xCortex-A57 MPCore freq. 0.45–1.1 GHz L2 2 MB
LITTLE	-	4xCortex-A7 MPCore freq. 250–600 MHz L2 512 KB	4xCortex-A53 MPCore freq. 450–850 MHz L2 1 MB

Table 4.3 shows the x86/Intel® targets used for the polar SC decoders evaluation while Table 4.4 summarizes the ARM® platforms. There are two platforms with Cortex-A15 cores. We decided to identify the ones from the Nvidia® Jetson TK1 board as the *A15-J* and the ones from the Hardkernel® ODROID-XU as the *A15-O*.

In this section, all the binaries have been compiled with the GNU compiler version 5.4 and with the following optimization flags: `-Ofast -funroll-loops`. All the proposed results are single threaded and the frequency boost is enabled on the Intel® CPUs. As a convention, performance of the intra-frame SIMD version of the SC decoder is represented in blue in the figures while performance of the inter-frame SIMD version is represented in red.

4.2.1.2 Performance and Energy Efficiency on Embedded CPUs

The objective and originality of this section is to explore different software and hardware parameters for the execution of a software SC decoder on ARM® architectures. For a software decoder implementation, many parameters can be explored, influencing performance and energy efficiency. The target rate and frame size are applicative parameters. The SIMDization strategies (intra-frame or inter-frame) and the features of decoders (generated or dynamic) are software parameters. Furthermore, the target architecture, its frequency and its voltage are hardware parameters. This study investigates the correlations between these parameters in order to better choose an efficient implementation for a given applicative purpose. The low-power general purpose ARM32 and ARM64 processor testbeds based on big.LITTLE architecture are selected as representative of modern multi-core and heterogeneous architectures.

The flexibility of the AFF3CT software enables to alter many parameters and turn many optimizations on or off, leading to a large amount of potential combinations. For the purpose of this study, computations are performed with 8-bit fixed-point data types, with all tree pruning optimizations activated. The main metric considered is the average amount of energy in Joules to decode one bit of information, expressed as $\mathsf{E_b} = (\mathsf{P} \times \mathcal{L})/(\mathsf{K} \times \mathsf{F})$ where P is the average power (Watts), $\mathcal{L}$ is the latency, K is the number of information bits and F is the number of frames decoded in parallel.

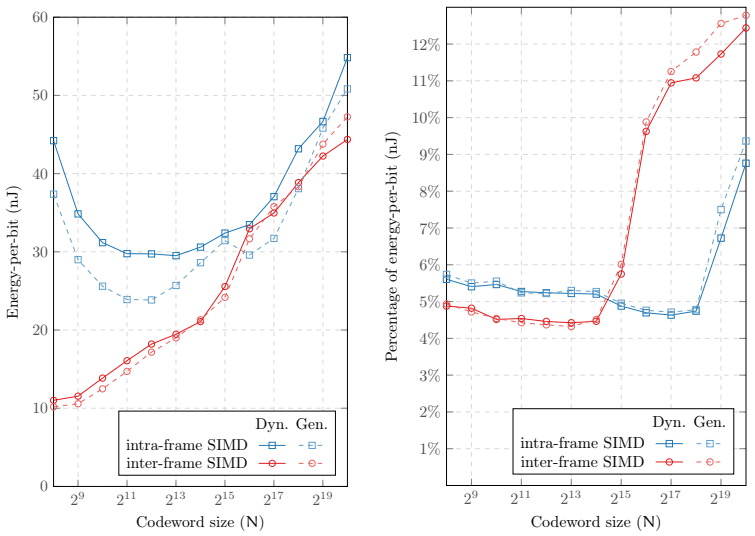
Table 4.5 – Characteristics for each cluster ($\mathcal{T}_\mathsf{i}$ is the information throughput), for dynamic SC decoders. $\mathsf{N} = 4096$, rate $\mathsf{R} = 1/2$. The RAM consumption is not included in P and in $\mathsf{E_b}$.

Cluster	**Freq.** (MHz)	**Impl.**	$\mathcal{L}$ (µs)	$\mathcal{T}_\mathsf{i}$ (Mb/s)	P (W)	$\mathsf{E_b}$ (nJ)
A7-O	450	seq.	655.0	3.1	0.117	37.8
		intra	158.0	13.0	0.123	9.5
		inter	1506.0	21.8	0.131	6.0
A53	450	seq.	966.0	2.1	0.062	29.0
		intra	203.0	10.1	0.070	7.0
		inter	1902.0	17.2	0.088	5.1
A15-O	1100	seq.	274.0	7.5	0.913	122.0
		intra	58.0	35.2	0.991	28.2
		inter	522.0	62.8	1.093	17.4
A57	1100	seq.	222.0	9.2	0.730	78.9
		intra	52.0	39.2	0.826	21.1
		inter	503.0	65.1	0.923	14.2
i7-4850HQ	3300	seq.	56.5	36.3	8.532	235.4
		intra	9.2	221.8	9.017	40.5
		inter	51.8	632.2	9.997	15.8

Table 4.5 gives an overview of the decoder behavior on different clusters and for various implementations. The code is always single threaded and only the 8-bit fixed-point decoders are considered. Indeed 32-bit floating-point versions are 4 times more energy consuming, on average. The sequential version is mentioned for reference only, as the throughput $\mathcal{T}_\mathsf{i}$ is much higher on vectorized versions. Generally the inter-frame SIMD strategy delivers better performance at the cost of a higher latency $\mathcal{L}$. Table 4.5 also compares the energy consumption of LITTLE and big clusters. The A53 consumes less energy than the A7-O. The A57 consumes less energy than the A15-O, respectively. This can be explained by architectural improvements brought by the more recent ARM64 platform. Despite the fact that the ARM64 is a development board, the ARM64

outperforms the ARM32 architecture. Finally we observe that the power consumption is higher for the inter-frame version than for the intra-frame one because it fills the SIMD units more intensively. One can note that the SIMD units consume more than the scalar pipeline. However, this is largely compensated by a much higher efficiency.

For comparison, the results for the Intel® Core™ i7-4850HQ, using SSE4.1 instructions (same vector length as ARM® NEON vectors) are also included. Even if the i7 is competitive with the ARM® big cores in terms of *energy-per-bit* ($\mathsf{E_b}$), these results show it is not well suited for the low power SDR systems because of its high power requirements.

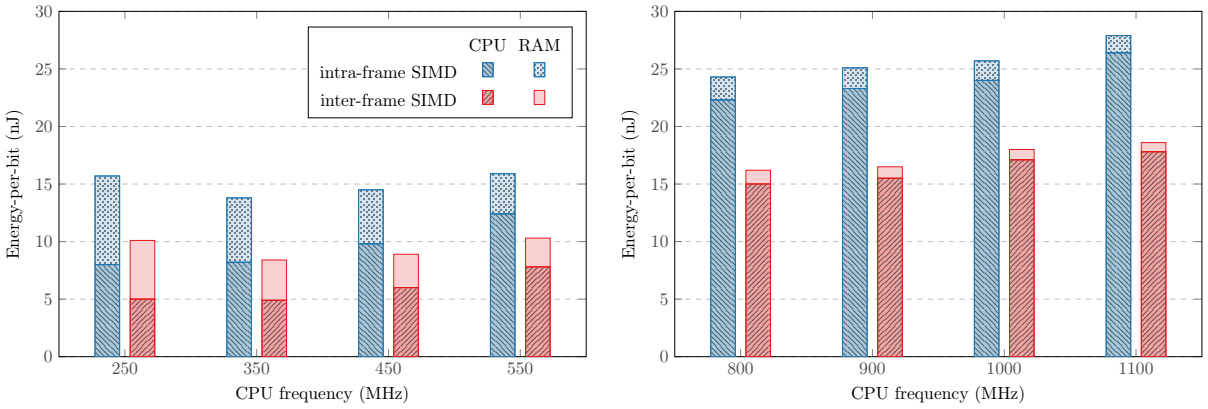


(a) Total (cluster + memory). (b) Memory only.

Figure 4.3 – Variation of the *energy-per-bit* for different frame sizes and implementations: intra-/inter-frame, dynamic and generated code, on A15-O @ 1.1 GHz and with a fixed code rate $\mathsf{R} = 1/2$.

Figure 4.3 shows the *energy-per-bit* consumption depending on the frame size N for the fixed rate $\mathsf{R} = 1/2$. In general, the energy consumption increases with the frame size. For small frame sizes (N from 2^8 to 2^{14}), the inter-frame SIMD outperforms the intra-frame SIMD. This is especially true for $\mathsf{N} = 2^8$ which has a low ratio of SIMD computations over scalar computations in the intra-frame version. As the frame size increases, the ratio of SIMD versus scalar computations increases as well. At some point around $\mathsf{N} = 2^{16}$ the intra-frame implementation begins to outperform the inter-frame one. Indeed, the data for the intra-frame decoder still fits in the CPU cache, whereas the data of the inter-frame decoder does not fit the cache anymore. In our case (8-bit fixed point numbers and 128-bit vector registers) the inter-frame decoders require 16 times more memory than the intra-frame decoders. Then, for the frame size $\mathsf{N} = 2^{20}$, both intra and inter-frame decoders now exceed the cache capacity. The RAM power consumption becomes more significant due to the increased number of cache misses causing RAM transactions. Considering those previous observations, it is more energy efficient to use inter-frame strategy for small frame sizes, whereas it is better to apply intra-frame strategy for larger frame sizes.

Figure 4.4 shows the impact of the frequency on the energy, for a given value of frame size $\mathsf{N} = 4096$ and a code rate $\mathsf{R} = 1/2$. On both A7-O and A15-O clusters, the supply voltage increases with the frequency from 0.946 V to 1.170 V. Results for the A7-O LITTLE cluster shows that the energy consumed by the system RAM is significant: At 250 MHz it accounts for half of the energy cost. Indeed, at low frequency, the long execution time due to the low throughput



(a) ARM® Cortex-A7 (ODROID-XU+E) (b) ARM® Cortex-A15 (ODROID-XU+E)

Figure 4.4 – Variation of the *energy-per-bit* (E_b) depending on the cluster frequency (dynamic code, intra-, inter-frame). $N = 4096$ and $R = 1/2$. Dark colors and light colors stand for CPU cluster and RAM energy consumption, respectively.

causes a high dynamic RAM refreshing bill. Therefore it is more interesting to use frequencies higher than 250 MHz. For this problem size and configuration, and from an energy-only point of view, the best choice is to run the decoder at 350 MHz. On the A15-O big cluster, the energy cost is driven by the CPU frequency, while the RAM energy bill is limited compared to the CPU.

Thus, the bottom line about energy versus frequency relationship is: On the LITTLE cluster it is more interesting to clock the CPU at high frequency (higher throughput and smaller latency for a small additional energy cost); On the big cluster, where the RAM consumption is less significant, it is better to clock the CPU at a low frequency.

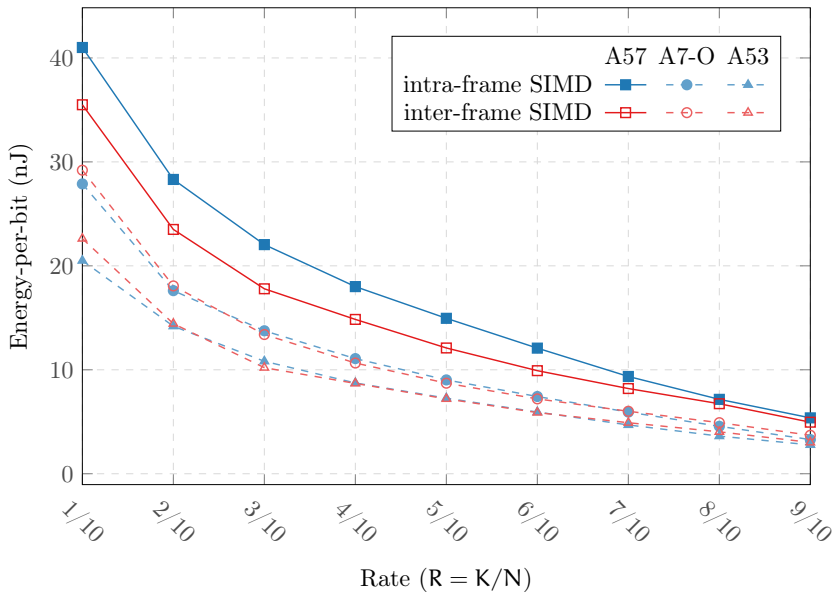


Figure 4.5 – Evolution of the *energy-per-bit* (E_b) for $N = 32768$ depending on the code rate $R = K/N$ (various impl.: intra-, inter-frame, code gen. on). Running on A7-O, A53 and A57 clusters @ 450MHz.

In Figure 4.5 the *energy-per-bit* cost decreases when the code rate increases. This is expected because there are many information bits in the frame when R is high. It makes the decoder more

energy efficient. With high rates, the SC decoding tree can be pruned more effectively. It makes the decoding process even more energy efficient. Figure 4.5 also compares the ARM® A7-O, A53 and A57 clusters for the same 450 MHz frequency (note: this frequency is not available on the A15-O). The LITTLE A7-O is more energy efficient than the big A57, and the LITTLE A53 is itself more energy efficient than the LITTLE A7-O ($\mathsf{E}_{\mathsf{b}_{\mathrm{A53}}} < \mathsf{E}_{\mathsf{b}_{\mathrm{A7\text{-}O}}} < \mathsf{E}_{\mathsf{b}_{\mathrm{A57}}}$).

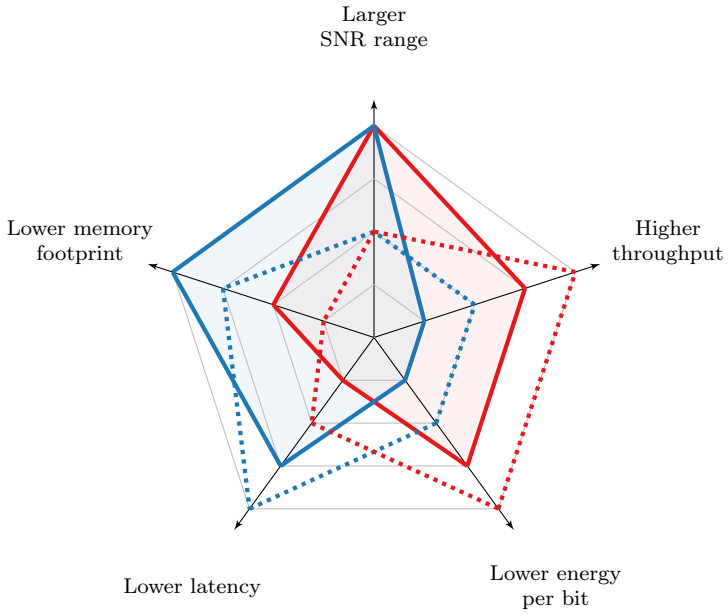


Figure 4.6 – Ranking of the different approaches along 5 metrics. In red, inter-frame vectorization performance and in blue, intra-frame performance. Solid color is for the dynamic versions and dotted is for the generated versions. Each version is positioned along each of the 5 axes and the best version for one axis is placed further from the center.

Figure 4.6 presents a qualitative summary of the characteristics of the different code versions, for intra-/inter-frame vectorization, generated or dynamic code. For instance, if the size of the memory footprint is an essential criterion, the dynamic intra-frame code exhibits the best performance.

To sum up, the dynamic implementations provides efficient trade-off between throughput, latency and energy depending on code length. It was demonstrated by previous benchmarks. Both implementations provide low-energy and low-power characteristics compared to previous works in the field on x86 processors [Sar+14a, Gia+14, Sar+14b, LLJ14, LLJ15, 4]. Whereas the throughput on a single processor core is reduced compared to x86 implementations, ARM® implementations must fulfil a large set of SDR applications with limited throughputs and where the power consumption matters. Finally, it is important to notice that multi-core implementations of the proposed ARM® decoders is still possible on these ARM® targets to improve the decoding throughputs.

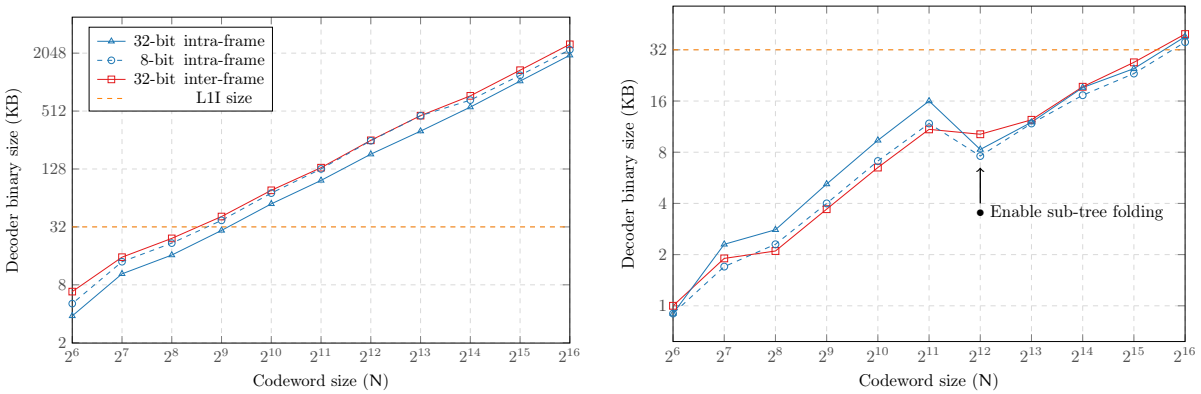
4.2.1.3 Source Code Compression for the Generated Decoders

For generated decoders, the corresponding binary size is linearly increasing with the codeword size N. Beyond a codeword size point which depends on the architecture and on the selected SIMD version, performance decreases due to L1I cache misses. Indeed, decoders are generated as straight-line code (no recursive calls), with all node computations put in sequence. This improves

performance for small to medium codeword size, up to the point where the compiled binary exceeds the L1I cache size. We mitigated this issue by reducing decoder binary sizes by applying two compression techniques: 1) in the generated code, we moved the buffer offsets from template arguments to function arguments. It enabled the compiler to factorize more function calls than before, 2) we implemented a sub-tree folding algorithm in the generator.

Table 4.6 – Binary code size (in KB) of the generated decoders depending on the number of bits N per frame.

Decoder	$N = 2^6$	$N = 2^8$	$N = 2^{10}$	$N = 2^{12}$	$N = 2^{14}$	$N = 2^{16}$
inter 32-bit, R = 1/2	1 (7)	2 (24)	7 (**77**)	9 (**254**)	19 (**736**)	**40** (**2528**)
inter 32-bit, R = 5/6	1 (4)	2 (19)	4 (**53**)	7 (**167**)	16 (**591**)	32 (**1758**)
intra 32-bit, R = 1/2	1 (4)	3 (16)	9 (**56**)	8 (**182**)	19 (**563**)	**38** (**1947**)
intra 32-bit, R = 5/6	1 (3)	3 (13)	6 (**38**)	7 (**126**)	20 (**392**)	27 (**1365**)
inter 8-bit, R = 1/2	1 (5)	2 (22)	7 (**72**)	8 (**252**)	17 (**665**)	**36** (**2220**)
inter 8-bit, R = 5/6	1 (4)	2 (18)	4 (**51**)	6 (**191**)	14 (**461**)	26 (**1555**)



(a) Without compression

(b) With compression

Figure 4.7 – Generated SC decoder binary sizes depending on the frame size (R = 1/2).

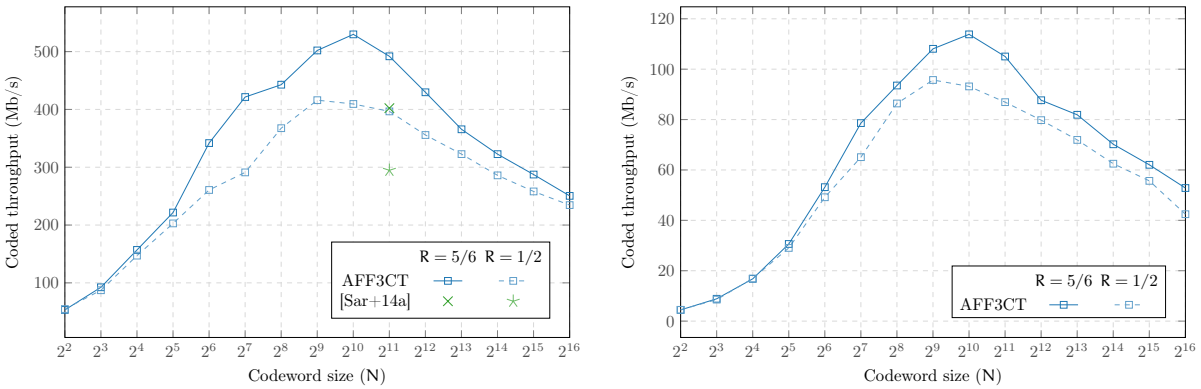
Table 4.6 and Figure 4.7 illustrate the binary code size of the decoders depending on N. The results which exceed the 32KB of the L1I cache are highlighted in bold font. A CPU with L1I = 32 KB is supposed, this is consistent with most of the current CPUs. Sub-tree folding is enabled starting from $N = 2^{12}$ because there is an overhead (at run-time) when using this technique. The source code is compiled with AVX instructions for the 32-bit decoders and with SSE4.1 instructions for the 8-bit decoders. AFF3CT decoder code sizes without compression are shown in parentheses: we can observe a huge improvement, until $N = 2^{14}$ the code size never exceeds the L1I cache anymore. In [Gia+16], authors report that they can't compile codes longer than $N = 2^{15}$. The proposed compression technique enables to exceed that limit. For instance, we were able to generate $N = 2^{20}$ decoders as shown in Figure 4.3.

4.2.1.4 Comparison with State-of-the-art SC Decoders

Dynamic Implementation Table 4.7 shows a performance comparison (throughput, latency) with the dynamic intra-frame decoder of [Gia+14]. On a x86 CPU, our dynamic decoder is 2.8 times faster than the state-of-the-art decoder. Even if we used a more recent CPU, the same set of instructions (SSE4.1) is applied and the frequencies are comparable.

Table 4.7 – Comparison of 8-bit fixed-point dynamic SC decoders (intra-frame SIMD). N = 32768 and R = 5/6.

Ref.	Platform	Freq. (GHz)	SIMD	$\mathcal{L}$ (µs)	$\mathcal{T}_i$ (Mb/s)
[Gia+14]	i7-2600	3.4	SSE4.1	135	204
[5]	i7-4850HQ	3.3	SSE4.1	47	**580**
[5]	A15-O	1.1	NEON	391	70
[5]	A57	1.1	NEON	374	73



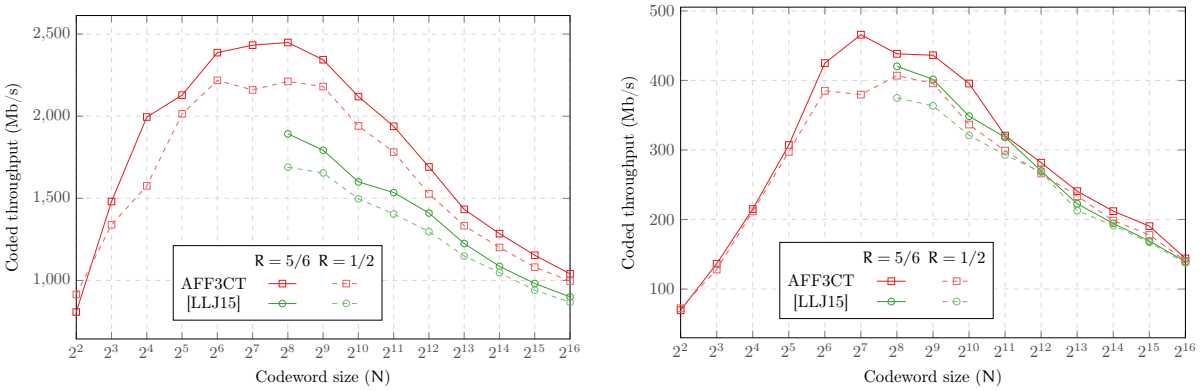
(a) Intel® Xeon™ E3-1225 (AVX SIMD)

(b) Nvidia® Jetson TK1 A15 (NEON SIMD)

Figure 4.8 – Performance comparison between two code rates of 32-bit floating-point decoding stages (intra-frame vectorization, generated SC decoders).

Table 4.8 – Comparing SC with a state-of-art generated software polar decoder, for different code sizes, using intra-frame SIMD. The two cross marks show state-of-the art performance results reported in [Sar+14a], for comparison. The AVX SIMD instructions are applied.

Ref.	(N, K)	Platform	$\mathcal{L}$ (µs)	$\mathcal{T}_i$ (Mb/s)
[Sar+14a]	(16384, 14746)	i7-2600	50	292
[4]		E3-1225	43	341
[Sar+14a]	(32768, 27568)	i7-2600	125	220
[4]		E3-1225	114	241
[Sar+14a]	(32768, 29492)	i7-2600	113	261
[4]		E3-1225	101	293



(a) Intel® Xeon™ E3-1225 (SSE4.1 SIMD) (b) Nvidia® Jetson TK1 A15 (NEON SIMD)

Figure 4.9 – Performance comparison between two code rates of 8-bit fixed-point decoding stages (inter-frame vectorization). Squares show AFF3CT generated SC decoders results. Circles show the "handwritten" implementation results from [LLJ15].

Generated Implementation Figure 4.8 shows AFF3CT intra-frame throughput on different architectures. Our generic framework performance outperforms previous works (between 10% and 25% higher). This is confirmed in Table 4.8 which compares AFF3CT with the state-of-the-art result samples for some specific code sizes reported in [Sar+14a]. The throughput of the inter-frame implementation is shown in Figure 4.9 for different architectures. Again, the results confirm that our generic approach overtakes handwritten code (also between 10% and 25% higher on x86). It worth mentioning that the decoder from [LLJ15] is not generated and can dynamically adapt to various frame sizes and code rates where our generated decoders cannot. To the best of our knowledge, there is no other generated implementation with the inter-frame SIMD strategy in the literature. By design, generated decoders are faster than the dynamic decoder (up to 20% on x86-based CPUs). But, it is less clear on ARM®-based CPUs.

4.2.2 Successive Cancellation List Decoders

Throughput and latency measurements of the dynamic SCL implementations (see Section 2.5.4) are detailed in this section. The proposed dynamic decoder implementations are compared with the previous software decoder implementations. Despite the additional levels of genericity and flexibility, the proposed software implementation is very competitive with its counterparts.

4.2.2.1 Experimentation Platforms

During our investigations, all the throughput and latency measurements have been obtained on a single core of an Intel® Core™ i5-6600K CPU (Skylake architecture with AVX2 SIMD) with a base clock frequency of 3.6 GHz and a maximum turbo frequency of 3.9 GHz. The description has been compiled on Linux with the C++ GNU compiler (version 5.4.0) and with the following options: `-Ofast -march=native -funroll-loops`.

4.2.2.2 Throughput and Latency of Adaptive SCL Decoders

Table 4.9 – Throughput and latency comparisons between floating-point (32-bit) and fixed-point (16-bit and 8-bit) Adaptive SSCL decoders. Code (2048,1723), $L = 32$ and 32-bit CRC (Gzip). $\mathcal{L}_{avg}$ is in µs and $\mathcal{T}_i$ is in Mb/s.

Decoder	Prec.	$\mathcal{L}_{worst}$ (µs)	3.5 dB		4.0 dB		4.5 dB	
			$\mathcal{L}_{avg}$	$\mathcal{T}_i$	$\mathcal{L}_{avg}$	$\mathcal{T}_i$	$\mathcal{L}_{avg}$	$\mathcal{T}_i$
PA-SSCL	32-bit	635	232.3	7.6	41.7	42.1	7.4	237.6
	16-bit	622	219.6	8.0	40.1	43.8	6.6	267.5
	8-bit	651	232.4	7.6	41.2	42.6	6.5	268.3
FA-SSCL	32-bit	1201	67.2	26.1	8.5	207.8	5.1	345.5
	16-bit	1198	68.7	25.6	7.7	225.7	4.3	408.7
	8-bit	1259	71.8	24.4	7.7	227.3	4.1	425.9

The property to easily change the list size of the SCL decoders enables the use of the FA-SSCL algorithm. With an unrolled decoder as proposed in [Sar+16], the fully adaptive decoder would imply to generate a fully unrolled decoder for each value of the list depth. In our approach, only one source code gives to the designer the possibility to run each variation of the SCL decoders. FA-SSCL algorithm is the key to achieve the highest possible throughput. In Table 4.9, maximum latency ($\mathcal{L}_{worst}$ in µs), average latency ($\mathcal{L}_{avg}$ in µs) and information throughput ($\mathcal{T}_i$ in Mb/s) are given. The FER performance of the 32-bit version of the PA/FA-SSCL decoders as well as the corresponding throughputs can be seen in Figure 3.4. The 16-bit and 8-bit implementations have similar decoding performance. Note that in 8-bit configuration only the $\texttt{REP}_{8-}$ nodes are used. The fixed-point implementation reduces, on average, the latency. In the high SNR region, the frame errors are less frequent. Therefore, the SCL algorithm is less necessary than in low SNR regions for Adaptive SCL algorithms. As the gain of fixed-point implementation benefits more to the SC algorithm than to the SCL algorithm, the throughput is higher in high SNR regions. With an 8-bit fixed-point representation of the decoder inner values, the achieved throughput in the case of the (2048,1723) polar code is about 425 Mb/s on the i5-6600K for an E_b/N_0 value of 4.5 dB. It corresponds to a FER of 5×10^{-8}. This throughput is almost 2 times higher than the throughput of the PA-SSCL algorithm. The highest throughput increase from PA-SSCL to FA-SSCL, of about 380%, is in the domain where the FER value is between 10^{-3} and 10^{-5}. It is the targeted domain for wireless communications like LTE or 5G standards. In these conditions, the throughput of FA-SSCL algorithm is about 227 Mb/s compared to 42 Mb/s for the PA-SSCL algorithm.

With Adaptive SCL algorithms, the worst case latency is the sum of the latency of each triggered algorithm. In the case of PA-SSCL with $L_{max} = 32$, it is just the sum of the latency of the SC algorithm, plus the latency of the SCL algorithm with $L = 32$. In the case of the FA-SSCL algorithm, it is the sum of the decoding latency of the SC algorithm and all the decoding latencies of the SCL algorithm for $L = 2, 4, 8, 16, 32$. This is the reason why the worst latency of the PA-SSCL algorithm is lower while the average latency. Consequently the average throughput is better with the FA-SSCL algorithm.

4.2.2.3 Comparison with State-of-the-art SCL Decoders

The throughput and latency of the proposed decoder compared to other reported implementations are detailed in Table 4.10. For all the decoders, all the available tree pruning optimizations are applied excluding the $\texttt{SPC}_{4+}$ nodes because of the performance degradation. Each decoder

Table 4.10 – Throughput and latency comparisons with state-of-the-art SCL decoders. 32-bit floating-point representation. Polar code (2048,1723), L = 32, 32-bit CRC.

Ref.	Target	Decoder	$\mathcal{L}_{\text{worst}}$ (µs)	$\mathcal{T}_{\text{i}}$ (Mb/s) 3.5 dB	4.0 dB	4.5 dB
[Sar+14c]	i7-2600	CA-SCL	23000	0.07	0.07	0.07
		CA-SSCL	3300	0.52	0.52	0.52
		PA-SSCL	≈ 3300	0.90	4.90	54.00
[She+16]	i7-4790K	CA-SCL	1572	1.10	1.10	1.10
[Sar+16]	i7-2600	CA-SCL	2294	0.76	0.76	0.76
		CA-SSCL	433	4.00	4.00	4.00
		PA-SSCL	≈ 433	8.60	33.00	196.00
[3]	i7-2600	CA-SCL	4819	0.37	0.37	0.37
		CA-SSCL	770	2.30	2.30	2.30
		PA-SSCL	847	5.50	31.10	168.40
		FA-SSCL	1602	19.40	149.00	244.30
[3]	i5-6600K	CA-SCL	3635	0.48	0.48	0.48
		CA-SSCL	577	3.00	3.00	3.00
		PA-SSCL	635	7.60	42.10	237.60
		FA-SSCL	1201	26.10	207.80	345.50

is based on a 32-bit floating-point representation. The polar code parameters are N = 2048, K = 1723 and the 32-bit GZip CRC is applied. The list size is L = 32.

The latency given in Table 4.10 is the worst case latency. The throughput is the average information throughput. The first version, CA-SCL, is the implementation of the CA-SCL algorithm without any tree pruning. As mentioned before the throughput of the proposed CA-SSCL decoder (2.3 Mb/s) is only halved compared to the specific unrolled CA-SSCL decoder (4.0 Mb/s) described in [Sar+16]. The proposed CA-SSCL decoder is approximately 4 times faster than the generic implementation (0.52 Mb/s) in [Sar+14c] and 2 times faster than the CA-SCL implementation (1.1 Mb/s) in [She+16] thanks to the implementation improvements detailed in Section 2.5.4. Furthermore, the proposed decoder exhibits a much deeper level of genericity and flexibility than the ones proposed in [Sar+14a, She+16]. Indeed, the following features are not enabled: the customization of the tree pruning, the 8-bit and 16-bit fixed-point representations of the LLRs, the puncturing patterns and the FA-SSCL algorithm.

When implemented on the same target (i7-2600), the proposed PA-SSCL is competitive with the unrolled PA-SSCL in [Sar+16], being only two times slower. This can be explained by the improvements concerning the CRC that are described in Section 2.5.4.1, especially the information bits extraction in the SC decoder. Finally, as mentioned before, the throughput of the proposed FA-SSCL significantly outperforms all the other SCL decoders (up to 345.5 Mb/s at 4.5 dB in 32-bit floating-point).

4.3 Turbo Decoders

In this section we propose to evaluate the turbo EML-MAP decoder presented in Section 2.6.2. The decoder throughput is benched on middle and high-end x86 CPUs. Then, the energy efficiency of the proposed decoder is studied. Finally, it is compared with the state-of-the-art decoders.

4.3.1 Experimentation Platforms

Table 4.11 – Specifications of the target processors.

	E5-2650	**i7-4960HQ**	**E5-2680 v3**
CPU	Intel® Xeon™ E5-2650	Intel® Core™ i7-4960HQ	Intel® Xeon™ E5-2680 v3
Cores/Freq.	8 cores, 2–2.8 GHz	4 cores, 2.6–3.8 GHz	12 cores, 2.5–3.3 GHz
Arch.	*Ivy Bridge* Q1'12	*Haswell* Q4'13	*Haswell* Q3'14
LLC	20 MB L3	6 MB L3	30 MB L3
TDP	95 W	47 W	120 W

The experiments have been conducted on three different x86-based processors detailed in Table 4.11. A mid-range processor (i7-4960HQ) is used for comparison with similar CPU targets in the literature [Hua+11, Zha+12, Wu+13]. The two high-end processors (E5-2650 and E5-2680 v3) are used for multi-threading benchmarking. Indeed, E5-2650 and E5-2680 v3 are potentially good candidates for C-RAN servers. Moreover, the code has been compiled with the GNU compiler (version 4.8) and with the `-Ofast -funroll-loops -msse4.1/-mavx2` options.

4.3.2 Throughput Performance on Multi-core CPUs

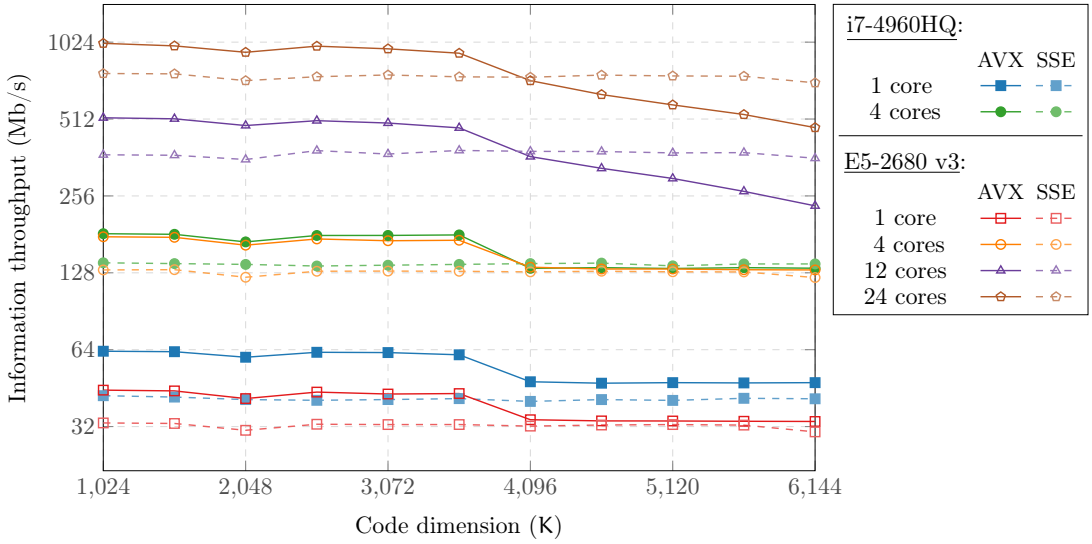


Figure 4.10 – Information throughput depending on K for various number of cores and SIMD instruction types. 6 iterations, 8-bit fixed-point.

Figure 4.10 shows the evolution of the information throughput depending on the code dimension K. This experiment was conducted on i7-4960HQ and E5-2680 v3 (both have *Haswell* architectures). The throughput tends to increase linearly with the number of cores (up to 24 cores) except in AVX mode where a performance drop can be observed when $K > 4096$. The reason is that the AVX instructions use vectors 2× wider than those used by SSE instructions. The inter-frame strategy loads twice the number of frames to fill these vectors. Thus, for $K > 4096$, in AVX, the memory footprint exceeds the L3 cache occupancy. Consequently, the performance is driven by the RAM bandwidth. Then, as K increases the number of RAM accesses increases. There is not enough memory bandwidth to feed all the cores. This explains the decreasing throughput for $K > 4096$, in AVX mode. Nonetheless, on E5-2680 v3 target, the throughput exceeds 1 Gbps for all codes with $K < 4096$.

4.3.3 Energy Efficiency on a Multi-core CPU

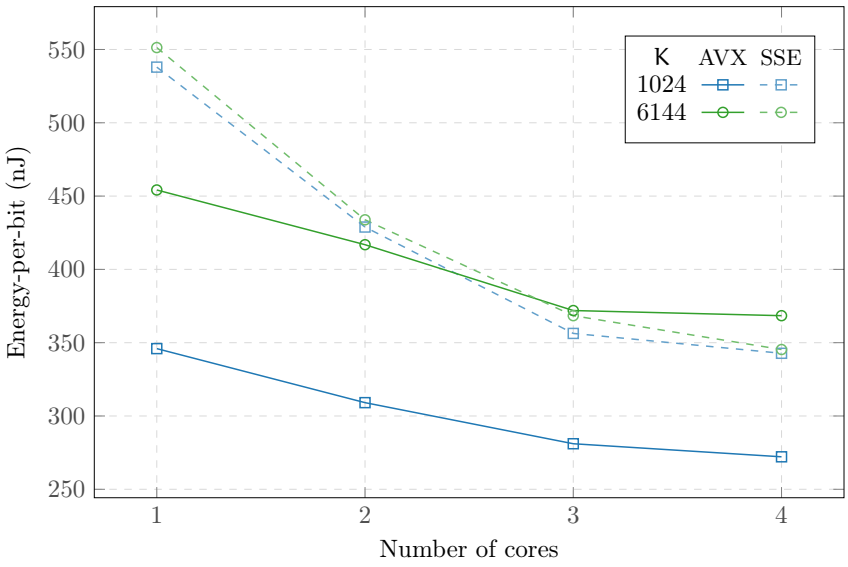


Figure 4.11 – *Energy-per-bit* ($\mathsf{E_d}$) depending on the number of cores and the instruction types. 6 iterations, 8-bit fixed-point.

Figure 4.11 shows the energy consumed by the processor to decode one information bit ($\mathsf{E_d}$) of the codes using SSE and AVX instructions on the i7-4960HQ CPU target. The throughput and power measurements were conducted on i7-4960HQ with the *Intel® Power Gadget* tool. For small codewords ($\mathsf{K} = 1024$) it is more energy efficient to resort to AVX. But this is not so clear on larger codewords ($\mathsf{K} = 6144$) since with 3/4 cores, the code using SSE outperforms the AVX one.

4.3.4 Comparison with State-of-the-art Turbo Decoders

Table 4.12 – Comparison of the proposed turbo decoder with the state-of-art. EML-MAP decoder ($\alpha = 0.75$). Code from the LTE standard, $\mathsf{K} = 6144$ and $\mathsf{R} = 1/3$. $\mathcal{NT}_\mathsf{i} = (\mathcal{T}_\mathsf{i} \times \mathsf{i})/(\text{Cores} \times 6)$.

Ref.	Platform	Cores	SIMD (length)	Pre. (bit)	Inter (level)	i	BER (at 0.7 dB)	FER (at 0.7 dB)	$\mathcal{L}$ (µs)	$\mathcal{T}_\mathsf{i}$ (Mb/s)	$\mathcal{NT}_\mathsf{i}$ (Mb/s)
[Zha+12]	X5670	6	16	8	6	3	6e-02	-	157	222.6	18.6
[Wu+13]	i7-3770K	4	8	16	4	6	-	1e-01	323	76.2	19.1
[6]	E5-2650	8	8	16	64	6	6e-06	5e-03	3665	107.3	13.4
[6]	i7-4960HQ	4	8	16	32	6	6e-06	5e-03	2212	88.9	22.2
[6]	2×E5-2680 v3	24	8	16	192	6	6e-06	5e-03	2657	443.7	18.5
[6]	E5-2650	8	16	8	128	6	5e-05	4e-02	3492	225.2	28.2
[6]	i7-4960HQ	4	16	8	64	6	5e-05	4e-02	2837	138.6	34.7
[6]	2×E5-2680 v3	24	16	8	384	6	5e-05	4e-02	3293	716.4	29.9
[LJ19]	2×E5-2680 v3	24	32	8	24	6	1e-03	3e-01	84	1735.0	72.3

Table 4.12 shows a performance comparison with related works. *SIMD* is the number of elements that can be computed in parallel in one SIMD instruction. *Pre.* is the precision of LLRs in bits. *Inter* is the number of frames computed in parallel. i is the number of turbo decoding iterations. *BER* and *FER* are the decoding performance of the decoder at 0.7 dB. $\mathcal{L}$ is the decoder latency. $\mathcal{T}_\mathsf{i}$ is the information throughput of the decoder and $\mathcal{NT}_\mathsf{i}$ is the normalized information

throughput: this metric considers 6 iterations on a single core. It enables to directly compare the throughput of the listed decoders. The FER performance of the proposed decoders are shown in Figure 3.5. One can note that using the 8-bit decoder leads to BER/FER degradations.

The variety of CPU targets and algorithmic parameters enables to show some global emerging trends. When comparing to similar CPU targets [Zha+12, Wu+13], the proposed implementation reaches similar or higher throughput (from 88.9 Mbps to 138.6 Mbps on i7-4960HQ target) at the price of an latency increase (from 2212 µs to 2837 µs) and additional memory footprint. The implementation from [LJ19] outperforms the throughput (up to 2,5 times higher) and the latency performance of our proposed decoder on the E5-2680 v3 target. The other works [Zha+12, Wu+13, LJ19] are based on the intra-frame SIMD strategy. It results in significantly reduced latencies compared to the proposed implementation. The intra-frame implementations use less memory than the inter-frame strategy. It is possible to take advantage of wider SIMD instructions for the selected code. It mainly explains the throughput performance difference between the proposed implementation and the implementation from [LJ19]. Moreover, the decoder has been specialized for the LTE interleaver. However in the LTE standard, the EML-MAP turbo decoder comes with a limited inherent parallelism of 8 (corresponding to the 8 trellis states). To fill the SIMD registers, the intra-frame implementations require to modify the algorithm to introduce more parallelism. This modification leads to non-negligible decoding performance losses. At 0.7 dB, none of the state-of-the-art implementations are able to match the reference decoding performance of the EML-MAP algorithm except our proposed implementation in 16-bit. Even considering our 8-bit implementation, none of the other works are reaching the same level of decoding performance.

To summarize, the proposed implementation comes with a throughput approaching to the best implementations ($\approx$ two times slower) while the latency is still very high compared to the intra-frame decoders. One of the main advantage of the proposed implementation is its flexibility. Indeed, it can be run on 32-bit floating-point and 16/8-bit fixed-point. There is also a unique source code description for the SSE, AVX, AVX-512 and NEON instructions. As it has been shown before, this is valuable because depending on the codeword size, the CPU and the number of cores assigned, the throughput and latency performance can be more interesting on one or the other of the SIMD engine. Moreover, the proposed implementation is able to reach the reference decoding performance in 16-bit. In 8-bit, there is contained decoding performance degradations compared to the other works in the literature.

4.4 FEC Software Decoders Hall of Fame

In this section three software decoders Hall of Fames (HoFs) are provided: 1) for the LDPC decoders in Table 4.13, 2) for the polar decoders in Table 4.14 and 3) for the turbo decoders in Table 4.15. The purpose of these HoFs is to see at a glance what has been achieved, what can be expected from current software decoders. Moreover, they enable to easily compare their respective characteristics. All the presented results, collected from the state-of-the-art research papers published in the field, consider a BPSK (Bit Phase-Shift Keying) modulation/demodulation and an AWGN (Additive White Gaussian Noise) channel. This Hall of Fame strives to present results as fairly as possible. For instance, early termination (*Early T.*) criteria are not taken into consideration while computing throughput, in order to compare raw performances using a consistent method. It remains possible, however, for typos/glitches/mistakes to have inadvertently made it to the scoreboard.

Table 4.13 – LDPC Software Decoders Hall of Fame.

	Work	Year	Hardware specifications					Code			Decoder parameters						Decoding perf.		Metrics		
			Platform	Arch.	TDP (W)	Cores (or SM)	SIMD (length)	Freq. (GHz)	(N, K)	Std.	# of Edges	Sche-duling	Early T.	Up. Rules	Pre. (bit)	F (iter)	i	$\mathcal{L}$ (µs)	$\mathcal{T}_c$ (Mb/s)	$\mathcal{NT}_c$ (Mb/s)	TNDC	E_a (nJ)
GPU-based	[WCW08]	2008	8800 GT	*Tesla*	105	7	16	1.50	(4096, 2048)	-	6144	BP-F	yes	SPA	32	1	6	467000	0.01	0.001	0.000006	105000000
	[Fal+09]	2009	8800 GTX	*Tesla*	176	8	16	1.35	(1908, 1696)	-	7632	BP-F	no	SPA	32	-	50	-	0.08	0.080	0.000500	2200000
	[FSS11]	2011	8800 GTX	*Tesla*	176	8	16	1.35	(8000, 4000)	-	24000	BP-F	no	SPA	8	-	50	-	10.10	10.100	0.058000	17426
	[Fal+11]	2011	Tesla C2050	*Fermi*	247	14	32	1.15	(64800, 21600)	DVB-S2	216000	BP-F	no	MS	8	16	30	13275	78.10	46.860	0.091000	5271
	[Wan+11a]	2011	GTX 470	*Fermi*	215	14	32	1.22	(1944, 972)	802.11n	6804	BP-F	yes	LSPA	32	300	50	57743	10.10	10.100	0.018000	21287
	[JCS11]	2011	GTX 285	*Tesla*	204	15	16	1.48	(2304, 1152)	802.16e	7296	BP-F	yes	SPA	32	1	15	1097	2.10	0.630	0.001800	323810
	[Cha+11]	2011	Tesla C1060	*Tesla*	200	15	16	1.30	(8000, 4000)	-	32000	BP-F	no	LSPA	32	1	50	8638	0.92	0.920	0.002900	217391
	[Wan+11b]	2011	GTX 470	*Fermi*	215	14	32	1.22	(2304, 1152)	802.16e	7296	BP-F	no	LSPA	32	224	10	10533	49.00	9.800	0.018000	21939
	[KM12]	2012	GTX 480	*Fermi*	250	15	32	1.40	(2048, 1723)	802.3an	12288	BP-F	yes	SPA	32	1	50	426	4.80	4.800	0.007100	52083
	[Fal+12]	2012	HD 5870	*Cypress*	188	20	20	1.20	(8000, 4000)	-	-	BP-F	no	MS	8	500	10	22222	180.00	36.000	0.075000	5222
	[Fal+12]	2012	Tesla C2050	*Fermi*	247	14	32	1.15	(8000, 4000)	-	-	BP-F	no	MS	8	500	10	20000	200.00	40.000	0.078000	6175
	[GNB12]	2012	Tesla C2050	*Fermi*	247	14	32	1.15	(16200, 8100)	DVB-T2	48599	BP-F	no	MS	8	128	50	26083	79.50	79.500	0.154000	3107
	[Li+13]	2013	GTX 580	*Fermi*	244	16	32	1.54	(2304, 1152)	802.16e	7296	BP-CL	no	MS	8	1024	5	3322	710.20	142.000	0.180000	1718
	[Wan+13]	2013	GTX TITAN	*Kepler*	250	14	192	0.84	(2304, 1152)	802.16e	7296	BP-F	yes	NMS	32	50	10	1266	304.20	60.800	0.027000	4112
	[Wan+13]	2013	GTX TITAN	*Kepler*	250	14	192	0.84	(2304, 1152)	802.16e	7296	BP-F	yes	NMS	32	6	10	207	66.80	13.400	0.006000	18657
	[LN14]	2014	GTX 660 Ti	*Kepler*	150	7	192	0.92	(8000, 4000)	-	24000	BP-F	no	SPA	8	12544	50	954100	105.20	105.200	0.085000	1426
	[LJC14]	2014	GTX 660	*Kepler*	140	5	192	0.98	(1944, 972)	802.11n	6804	BP-HL	no	OMS	8	16384	10	34362	926.90	185.400	0.049000	755
	[Lai+16]	2016	GTX 470	*Fermi*	215	14	32	1.22	(1944, 972)	802.11n	6804	BP-PL	no	MS	32	256	10	9739	51.10	10.200	0.019000	21078
	[KK17b]	2017	GTX TITAN X	*Pascal*	250	28	128	1.42	(1944, 972)	802.11n	6804	BP-F	no	MS	32	1	10	2	913.00	182.600	0.036000	1369
	[KK17a]	2017	GTX TITAN X	*Pascal*	250	28	128	1.42	(1944, 972)	802.11n	6804	BP-F	no	MS	32	28	10	33	1660.00	332.000	0.065000	753
	[Kun18]	2018	GTX TITAN Xp	*Pascal*	250	30	128	1.58	(64800, 21600)	DVB-S2	216000	BP-F	yes	OMS	32	1	50	405	160.00	160.000	0.026000	1563
CPU-based	[Fal+08]	2008	CELL/BE	*CELL*	200	6	16	3.30	(1248, 624)	802.16e	-	BP-F	no	MS	8	96	25	3653	32.80	16.400	0.052000	6098
	[FSS11]	2011	CELL/BE	*CELL*	200	6	4	3.30	(1024, 512)	-	3072	BP-F	no	SPA	32	24	50	1719	14.30	14.300	0.181000	13986
	[FSS11]	2011	2xE5530	*Nehalem*	160	8	4	2.40	(8000, 4000)	-	24000	BP-F	no	SPA	32	1	50	13115	0.61	0.610	0.007900	262295
	[Zha+11]	2011	CELL/BE	*CELL*	200	8	16	3.20	(960, 480)	802.16e	-	BP-F	no	OMS	8	1	15	74	13.00	3.900	0.009500	51282
	[GNB12]	2012	i7-950	*Nehalem*	130	4	16	3.06	(16200, 8100)	DVB-T2	48599	BP-F	no	MS	8	128	50	113934	18.20	18.200	0.093000	7143
	[Pan+13]	2013	i7-3960X	*S. Bridge*	130	6	16	3.30	(9216, 4608)	CMMB	27648	BP-F	yes	NMS	8	12	10	1202	92.00	18.400	0.058000	7065
	[HNH13]	2013	i7-2600K	*S. Bridge*	130	4	16	3.40	(524280, 262140)	802.11n	-	BP-L	no	OMS	8	1	5	17420	30.10	3.000	0.055000	31667
	[GB13]	2013	Cortex-A9	*ARMv7*	≈ 4	4	16	1.60	(16200, 8100)	DVB-T2	48599	BP-F	no	MS	8	128	20	592457	3.50	1.400	0.014000	2857
	[Deb+16b]	2016	i7-4960HQ	*Haswell*	47	4	8	3.40	(2304, 1152)	802.16e	7296	LP-F	no	ADMM	32	4	8	1511	6.10	0.980	0.009000	47959
	[Deb+16a]	2016	i7-4960HQ	*Haswell*	47	4	8	3.40	(2304, 1152)	802.16e	7296	LP-HL	no	ADMM	32	32	100	13755	5.40	10.800	0.099000	4352
	[LJ16]	2016	i7-4960HQ	*Haswell*	47	4	32	3.40	(2304, 1152)	802.16e	7296	BP-HL	yes	NMS	8	128	50	1359	217.00	217.000	0.500000	217
	[LJ17]	2017	i7-5650U	*Broadwell*	≈ 10	2	32	3.00	(2304, 1152)	802.16e	7296	BP-HL	yes	OMS	8	2	10	12	385.00	77.000	0.401000	123
	[Gra19]	2019	2xEPYC 7351	*Zen*	340	32	16	2.40	(64800, 32400)	DVB-S2	226799	BP-HL	yes	OMS	8	512	20	18432	1800.00	720.000	0.586000	472
	[Xu+19]	2019	Gold 6154	*Skylake*	200	18	64	3.00	(9126, 8448)	5G	-	BP-HL	yes	OMS	8	18	10	31	4892.40	978.500	0.283000	204
	This work	2020	Platinum 8168	*Skylake*	205	24	32	2.70	(2304, 1152)	802.16e	7296	BP-HL	yes	NMS	16	768	50	2637	671.04	671.040	0.323600	305
	This work	2020	EPYC 7452	*Zen 2*	155	32	16	2.35	(2304, 1152)	802.16e	7296	BP-HL	yes	NMS	16	512	50	1368	862.08	862.080	0.716500	180

Table 4.14 – Polar Software Decoders Hall of Fame.

	Work	Year	Hardware specifications					Code		Decoder parameters				Decoding perf.		Metrics		
			Platform	Arch.	TDP (W)	Cores (or SM)	SIMD (length)	Freq. (GHz)	N	R	Algorithm	Pre. (bit)	F (inter)	i/L	$\mathcal{L}$ (µs)	$\mathfrak{T}_i$ (Mb/s)	$\mathcal{NT}_i$ (Mb/s)	TNDC	E_d (nJ)
GPU-based	[Gia+16]	2016	Tesla K20c	*Kepler*	225	13	192	0.71	4096	0.90	SSC	32	832	1	9400	1043.00	1043.00	0.5890	216
	[LL16]	2016	Tesla K20c	*Kepler*	225	13	192	0.71	256	0.50	SSC	32	-	1	-	395.00	395.00	0.2230	570
	[Cam+17]	2017	GTX 980 Ti	*Maxwell*	250	22	128	1.00	4096	0.50	BP+CA-SCL	32	5	32	1000000	0.01	0.32	0.0001	781250
	[Han+17]	2017	GTX 980	*Maxwell*	165	16	128	1.17	4096	0.50	SCL	32/16	1310	32	111900	24.00	768.00	0.3205	215
	[Han+17]	2017	GTX TITAN X	*Maxwell*	250	24	128	1.00	4096	0.50	SCL	32/16	1918	32	126700	31.00	992.00	0.3229	252
CPU-based	[Gia+14]	2014	i7-2600	*S. Bridge*	95	4	8	3.40	32768	0.84	SSC	32	1	1	223	123.70	123.70	4.5480	768
	[Gia+14]	2014	i7-2600	*S. Bridge*	95	4	16	3.40	32768	0.84	SSC	8	1	1	135	203.60	203.60	3.7430	467
	[LLJ14]	2014	Cortex-A9	*ARMv7*	≈ 3	4	16	1.30	32768	0.90	SSC	8	16	1	16852	28.00	28.00	1.3460	107
	[Sar+14c]	2014	i7-2600	*S. Bridge*	95	4	8	3.40	2048	0.84	CA-SSCL	32	1	32	3300	0.52	16.64	0.5882	5709
	[Sar+14a]	2014	i7-2600	*S. Bridge*	95	4	8	3.40	32768	0.84	SSC	32	1	1	125	219.80	219.80	8.0810	432
	[LLJ15]	2015	i7-4960HQ	*Haswell*	47	4	16	3.60	32768	0.90	SSC	8	16	1	337	1400.00	1400.00	24.3060	34
	[4]	2015	E3-1225	*S. Bridge*	95	4	8	3.10	32768	0.84	SSC	32	1	1	114	241.00	241.00	9.7180	394
	[4]	2015	E3-1225	*S. Bridge*	95	4	16	3.10	32768	0.83	SSC	8	16	1	370	1180.00	1180.00	23.7900	81
	[Sar+16]	2016	i7-2600	*S. Bridge*	95	4	8	3.40	2048	0.84	CA-SSCL	32	1	32	433	4.00	128.00	4.7059	742
	[Gia+16]	2016	i7-4770S	*Haswell*	64	4	32	3.10	32768	0.84	SSC	8	1	1	31	886.00	886.00	8.9310	73
	[Gia+16]	2016	Cortex-A9	*ARMv7*	≈ 3	4	16	1.70	32768	0.90	SSC	8	1	1	361	81.70	81.70	3.0030	37
	[5]	2016	i7-4850HQ	*Haswell*	47	4	16	3.30	32768	0.83	SSC	8	1	1	47	580.00	580.00	10.9840	81
	[5]	2016	Cortex-A57	*ARMv8*	≈ 2	2	16	1.10	32768	0.83	SSC	8	1	1	374	73.00	73.00	4.1480	27
	[She+16]	2016	i7-4790K	*Haswell*	88	4	8	4.00	2048	0.84	SCL	32	1	1	1573	1.10	35.10	1.0938	2514
	[LLJ18]	2018	i7-4960HQ	*Haswell*	47	4	32	3.60	32768	0.84	SSCAN	32	1	1	56	490.00	490.00	4.2535	96
	[LLJ18]	2018	i7-4960HQ	*Haswell*	47	4	32	3.60	32768	0.84	SSCAN	32	32	1	1601	550.00	550.00	4.7743	85
	[3]	2019	i5-6600K	*Skylake*	91	4	32	3.90	2048	0.84	CA-SSCL	8	1	32	577	3.00	96.00	0.7692	948

Table 4.15 – Turbo Software Decoders Hall of Fame.

	Work	Year	Hardware specifications					Code			Decoder parameters			Decoding performances				Metrics		
			Platform	Arch.	TDP (W)	Cores (or SM)	SIMD (length)	Freq. (GHz)	K	R	Std.	Algorithm	Pre. (bit)	F (inter)	i	BER (at 0.7 dB)	FER	$\mathcal{L}$ (µs)	$\mathfrak{T}_i$ (Mb/s)	$\mathcal{NT}_i$ (Mb/s)	TNDC	E_d (nJ)
GPU-based	[WSC10]	2010	Tesla C1060	*Tesla*	200	15	16	1.30	6144	1/3	LTE	ML-MAP	32	100	5	1e-04	-	76800	8.0	6.7	0.021	29851
	[Wu+11]	2011	GTX 470	*Fermi*	215	14	32	1.22	6144	1/3	LTE	ML-MAP	32	100	5	4e-05	-	20827	29.5	24.6	0.045	8740
	[CS12]	2012	Tesla C2050	*Fermi*	247	14	32	1.15	11918	1/3	-	L-MAP	32	32	5	-	-	108965	3.5	2.9	0.0057	85172
	[YC12]	2012	9800 GX2	*Tesla*	197	16	16	1.50	6144	1/3	LTE	ML-MAP	32	1	5	1e-02	-	3072	2.0	1.7	0.0043	115882
	[Liu+13]	2013	GTX 550 Ti	*Fermi*	116	6	32	1.80	6144	1/3	LTE	EML-MAP	32	1	6	1e-02	-	72	85.3	85.3	0.247	1360
	[Che+13]	2013	GTX 580	*Fermi*	244	16	32	1.54	6144	1/3	LTE	ML-MAP	32	1	6	3e-04	-	1660	3.7	3.7	0.0047	63946
	[Xia+13]	2013	GTX 480	*Fermi*	250	15	32	1.40	6144	1/3	LTE	EML-MAP	32	1	6	-	-	50	122.8	122.8	0.183	2036
	[Wu+13]	2013	GTX 680	*Kepler*	195	8	192	1.01	6144	1/3	LTE	EML-MAP	32	16	6	-	1e-02	2657	37.0	37.0	0.024	5270
	[Zha+14b]	2014	Tesla K20c	*Kepler*	225	13	192	0.71	6144	1/3	LTE	ML-MAP	32	1	5	1e-04	-	1097	5.6	4.7	0.0026	47872
	[Li+14]	2014	GTX 580	*Fermi*	244	16	32	1.54	6144	1/3	LTE	BR-SOVA	8	4	5	2e-02	-	192	127.8	106.5	0.135	2291
	[Li+16]	2016	GTX 680	*Kepler*	195	8	192	1.01	6144	1/3	LTE	EML-MAP	32	1	7	9e-03	-	817	8.2	9.6	0.0062	20313
	[Li+16]	2016	GTX 680	*Kepler*	195	8	192	1.01	6144	1/3	LTE	FPTD	32	1	36	9e-03	-	403	18.7	-	-	-
CPU-based	[Hua+11]	2011	i7-960	*Nehalem*	130	1	8	3.20	1008	1/3	LTE	ML-MAP	16	1	8	3e-03	7e-02	138	7.3	9.7	0.380	13402
	[Zha+12]	2012	X5670	*Westmere*	95	6	16	2.93	5824	1/3	LTE	EML-MAP	8	6	3	6e-02	-	157	222.6	111.3	0.396	854
	[Wu+13]	2013	i7-3770K	*I. Bridge*	77	4	8	3.50	6144	1/3	LTE	EML-MAP	16	4	6	-	1e-01	323	76.2	76.2	0.680	1011
	[6]	2016	E5-2650	*I. Bridge*	95	8	8	2.50	6144	1/3	LTE	EML-MAP	16	64	6	6e-06	6e-03	3665	107.3	107.3	0.669	885
	[6]	2016	i7-4960HQ	*Haswell*	47	4	8	3.20	6144	1/3	LTE	EML-MAP	16	32	6	6e-06	6e-03	2212	88.9	88.9	0.868	527
	[6]	2016	2×E5-2680 v3	*Haswell*	240	24	8	2.50	6144	1/3	LTE	EML-MAP	16	192	6	6e-06	6e-03	2657	443.7	443.7	0.924	541
	[6]	2016	E5-2650	*I. Bridge*	95	8	16	2.50	6144	1/3	LTE	EML-MAP	8	128	6	8e-05	5e-02	3492	225.2	225.2	0.704	422
	[6]	2016	i7-4960HQ	*Haswell*	47	4	16	3.20	6144	1/3	LTE	EML-MAP	8	64	6	8e-05	5e-02	2837	138.6	138.6	0.677	339
	[6]	2016	2×E5-2680 v3	*Haswell*	240	24	16	2.50	6144	1/3	LTE	EML-MAP	8	384	6	8e-05	5e-02	3293	716.4	716.4	0.746	335
	[LJ19]	2019	2×E5-2680 v3	*Haswell*	240	24	32	2.50	6144	1/3	LTE	EML-MAP	8	24	6	1e-03	3e-01	84	1735.0	1735.0	0.904	138

All the entries are sorted by platform type (GPU and CPU) and chronologically. For each HoF table, a column is dedicated to the *Hardware specifications*. Indeed, it can have a huge impact on the decoding performance depending on if you are using a low power ARM® CPU or a high end HPC GPU for instance. Then the next column aims to give some insights on the *Code* used in the decoding process. Again, decoding a short code generally results in low latencies performance when decoding a big code generally leads to higher latencies. The *Decoder parameters* are given in a dedicated column. Those parameters characterize the decoding process used in the experiments. The *Decoding performances* (or *Decoding perf.*) column shows the reported performances in terms of latency and throughput (and exceptionally the BER and FER in the turbo HoF). For the LDPC HoF, the throughput considers the codeword size (coded throughput $\mathcal{T}_c$, N bits) whereas in the polar and turbo HoFs the information throughputs are presented ($\mathcal{T}_i$ considering K bits). Generally speaking, the throughput ($\mathcal{T}$) can be deduced from the number of bits B in the codeword (B can be K or R depending on if we are considering information or coded throughput), the number of frames decoded in parallel (F) and the latency ($\mathcal{L}$):

$$\mathcal{T} = (B \times F)/\mathcal{L}. \tag{4.1}$$

In many cases, the raw performances are hard to directly compare. Indeed, the number of iterations (i) or the number of lists (L) can vary from a work to another one. This is why we proposed some *Metrics* in the last column. Those metrics aim to facilitate the comparison between the results. The *normalized throughput* ($\mathcal{NT}$) is different for each HoF. But, the idea is to normalize the throughput with a representative number of iterations/lists (I). The normalized throughput can be expressed as follows:

$$\mathcal{NT} = (\mathcal{T} \times i)/I, \tag{4.2}$$

for the LDPC HoF $I = 50$, for the turbo HoF $I = 6$ and for the polar HoF $I = 1$. The *Throughput under Normalized Decoding Cost* (*TNDC*) is a metric proposed in [Yin+12]. The general idea is to see how much the hardware components are stressed (higher TNDC is better). In the initial paper, the TNDC was only taking care of the frequency and the number of cores. In this thesis we refined the model by adding the SIMD length as this is a key for performance in channel decoding algorithms:

$$\text{TNDC} = \mathcal{NT}/(\text{Freq.} \times \text{Cores} \times \text{SIMD}). \tag{4.3}$$

The last metric is the *decoding energy* (E_d), this is the energy cost of the proposed implementation (lower is better):

$$E_d = (\text{TDP}/\mathcal{NT}) \times 10^3. \tag{4.4}$$

For the TNDC and the decoding energy, the normalized throughput is considered instead of the raw throughput. So one can compare metrics with each other. One can note that there is additional information given by the colors in the different tables. The definition of the colors is:

- blue: only one core of the CPU is used, in the TNDC computation one core is considered, in E_d the entire TDP is used,
- green: not including the memory data transfer time between the CPU and the GPU, in real life those transfers occur and the impact on the real latency can be significant,
- orange: following the formula, the throughput should be lower but the authors performed a specific data transfers overlapping with CUDA streams to reach higher throughput,
- purple: the inter-frame level has been deduced from the throughput and the latency.

4.5 SCMA Demodulators

In this section, the effects of the various SCMA demodulator optimizations considered in Section 2.7 are investigated. Energy efficiency, power consumption, throughput and latency are discussed.

4.5.1 Experimentation Platforms

Table 4.16 – Specifications of the target processors.

	i7-6700HQ	**7120P**	**A57**
CPU	Intel® Core™ i7-6700HQ	Intel® Xeon Phi™ 7120P	ARM® Cortex-A57 (Nvidia® Jetson TX1)
Cores/Freq.	4 cores, 2.6–3.5 GHz	64 cores, 1.24–1.33 GHz	4 cores, 1.91 GHz
Arch.	*Ivy Bridge* Q1'12	*Knights Corner* Q2'13	*ARMv8* Q1'15
LLC	6 MB L3	30.5 MB L2	2 MB L2
TDP	45 W	300 W	15 W

Energy efficiency is of interest in the design of C-RAN servers. It is determined by the rate of computation that can be achieved by a processor. Joint optimization of the throughput and energy consumption is a main goal of system designers. Energy optimization can significantly reduce the cost of cloud services while it can contribute to decrease the emission of greenhouse gases. Power utilization is also important because improved performance per Watt is useful to limit power demands. This section explores the power, energy efficiency and throughput of the various message passing algorithms suggested in this work. Tests have been conducted on three platforms running the Ubuntu Linux operating system. The three systems are : 1) an Intel® Core™ i7-6700HQ processor with AVX instructions (256-bit SIMD) and four physical cores using 2-way Simultaneous Multi-Threading (SMT or Intel® Hyper-Threading technology) running at nominal frequency of 2.6 GHz, 2) an ARM® Cortex-A57 with NEON instructions (128-bit SIMD) and four cores (no SMT) running at 2.0 GHz and 3) an Intel® Xeon Phi™ 7120P with KNCI instructions (512-bit SIMD) and 61 cores using 4-way SMT and running at 1.2 GHz. On the i7-6700HQ and the Cortex-A57 targets the GNU compiler (version 5.4) has been used while on the Xeon Phi™ co-processor the Intel compiler (version 17) has been used.

4.5.2 Throughput, Latency and Energy Efficiency on Multi-core CPUs

Table 4.17 shows the comparison of throughput, latency, power consumption and energy of different decoding algorithms that are executed on the three platforms to decode 768 Million bits. The average power and energy consumption measured on the Core™ i7 processor were obtained with the *turbostat* software [1] which exploits the Intel® performance counters in Machine Specific Registers (MSRs) to monitor CPU and RAM utilizations. However, in the case of ARM® and Xeon Phi™ platforms, external current sensors were used to measure the energy and power consumptions.

i7-6700HQ Platform During the evaluation process, 8 threads are run on the i7-6700HQ platform. The baseline implementation of MPA with level 3 (`-O3`) optimization of the GNU compiler reaches 3.51 Mbps by assigning all four physical cores of the processor (SMT on).

1. turbostat: https://github.com/torvalds/linux/tree/master/tools/power/x86/turbostat

Table 4.17 – MPA throughput, latency, power and energy characteristics over 768 Million bits [2].

	Algorithm & SIMD	Optim. Level	$\mathcal{L}$ (s)	$\mathcal{T}$ (Mb/s)	P (W)	E_b (µJ)
i7-6700HQ	E-MPA+AVX	`-Ofast`	81.4	75.46	40.02	0.53
	MPA+AVX	`-Ofast`	90.6	67.83	40.53	0.59
	Log-MPA	`-Ofast`	595.3	10.31	35.11	3.40
	Log-MPA	`-O3`	960.0	6.37	33.11	5.17
	MPA	`-Ofast`	412.9	14.85	33.01	2.22
	MPA	`-O3`	1745.5	3.51	35.00	9.94
7120P	E-MPA+KNCI	`-O2`	1634.0	114.60	198.00	1.73
	MPA+KNCI	`-O2`	2258.8	82.32	198.00	2.39
	Log-MPA	`-O2`	3490.9	53.38	184.00	3.43
	MPA	`-O2`	5120.0	36.09	196.00	5.36
A57	E-MPA+NEON	`-Ofast`	200.5	15.30	7.93	0.51
	MPA+NEON	`-Ofast`	365.7	8.40	7.56	0.90
	Log-MPA	`-Ofast`	650.1	4.70	6.99	1.48
	Log-MPA	`-O3`	1024.0	3.01	6.99	2.33
	MPA	`-Ofast`	752.9	4.07	7.18	1.76
	MPA	`-O3`	1920.0	1.60	6.99	4.37

Log-MPA algorithms improves the performance to 6.37 Mbps thanks to the deletion of the exponential calculations, still in `-O3`. However, using the fast math libraries (`-Ofast`) and the loop optimizations from Section 2.7.2 increases the throughput to 14.85 Mbps for MPA and to 10.31 Mbps for log-MPA. It is important to observe that MPA outperforms the log-MPA with the fast math libraries and more aggressive optimizations, without compromising on the bit error rate performance. This is because log-MPA induces inefficient data accesses due to the messages passed from resources to users. Using the AVX and SSE SIMD ISAs reduces the branch mispredictions and the cache misses (see Section 2.7.1). Consequently, the throughput is increased to 67.83 Mbps in MPA and to 75.46 Mbps for the E-MPA where the Ψ' estimated exponentials from (1.24) are performed. These results confirm significant throughput gains for the proposed implementations, while the energy consumption is reduced. AVX instructions increase the average power consumption of MPA and log-MPA from 35 to 40 Watts but throughput and latency are improved by much larger factors. It means that the overall energy consumption have been decreased with AVX instructions.

7120P Platform The Xeon Phi™ Knights Corner (KNC) [Chr12] benefits from the ability to execute four hardware threads per core, while having 61 cores and 512-bit SIMD registers. In this case, 244 threads are run to handle the MPA decoding task. Despite these benefits, the Xeon Phi™ Knights Corner suffers from two main disadvantages: 1) the KNC instruction set diversity is reduced compared to AVX or AVX-512 ISAs and 2) the cores frequency is relatively low in order to keep reasonable power consumption and limits the heat dissipation. As an example of missing instruction, the KNCI ISA does not offer coalesced division for floating-point numbers. Beside those limitations, the E-MPA+KNCI exhibits the highest throughput among the three mentioned platforms (up to 114.60 Mbps). However, it consumes almost three times more energy per bit compared to the ARM®-based implementations. On the Intel® ICPC compiler, the best performances are obtained using the `-O2` optimization level. Enabling the `-O3` optimization level and the fast math library does not lead to higher throughputs.

A57 Platform On the *Nvidia® Jetson TX1* platform, the throughput difference caused by the fast math libraries of the GNU compiler is still visible for MPA and log-MPA algorithms. With level three optimization (`-O3`), MPA and log-MPA run at 1.60 Mbps and 3.01 Mbps, respectively. When using fast math libraries (`-Ofast`) the throughputs increase to 4.07 and 4.70 Mbps. It should be noted that the four physical cores of the ARM® platform were utilized for those tests. Power consumption and energy used per decoded bit are lower on the ARM® platform than on the Intel® processors. The low power consumption of the ARM® platform notably comes at the cost of less powerful floating-point arithmetic units (see MPA+NEON and E-MPA+NEON in Table 4.17). Eliminating the exponential computations almost doubled the performance in E-MPA (15.30 Mbps) as compared to MPA+NEON (8.40 Mbps). It shows the limits of low power processors when many exponentials have to be calculated. Nevertheless, by using E-MPA, the ARM® low power processors can be a good candidate for implementation of SCMA decoders on C-RAN servers as it enables significant energy savings.

Considering the energy consumed per decoded bit ($\mathsf{E_b}$), the SIMD algorithms have a higher energy efficiency. The processor resources are well stressed and the power does not increase too much. Among the obtained results, the Xeon Phi™ achieves the best throughput while the Cortex-A57 has the lowest energy consumption. If the number of users in the cloud increases, then the presented results are scalable up to the number of processing units dedicated to them.

4.6 Analysis of the Simulator Performance

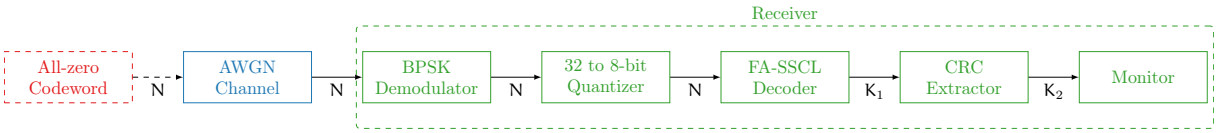


Figure 4.12 – AFF3CT simulator chain.

In this section we propose to evaluate the simulator performance over a representative simulation chain. This chain is illustrated in Figure 4.12. The transmitter is simplified and only all-zero codewords are generated. This technique enables to bench only the AWGN channel and the receiver tasks. The AWGN channel is presented in Section 1.2 and the vectorized implementation from the Section 2.3.1 is applied. The SNR is set to $\mathsf{E_b/N_0} = 4.5$ dB. The BPSK demodulator implementation is also vectorized and its implementation has not been detailed in the manuscript. Indeed, it is trivial and can be resumed to the multiplication of the AWGN channel output by a constant factor ($\mathbf{l} = \mathbf{y} \times \frac{2}{\sigma^2}$). Then, the demodulator 32-bit floating-point output data is converted in a 8-bit fixed-point representation thanks to the quantizer task (see Section 2.3.2). The decoder is the 8-bit polar FA-SSCL presented in Section 1.3.3.5 an in Section 2.5.4. A $\mathsf{N} = 2048$ and $\mathsf{K_1} = 1755$ polar code with $\mathsf{L} = 32$ and a 32-bit CRC (GZip `0x04C11DB7`) is simulated (see Figure 3.4 for the error rate). This decoder has been chosen because it represents one of the best optimized decoder of this thesis work. After the decoding process, the 32 CRC bits are extracted from the polar codeword and $\mathsf{K_2} = 1723$ bits are returned. This operation consists in copying the $\mathsf{K_2}$ first bits of the $\mathsf{K_1}$ decoder output bits. Finally, the monitor checks if all the bits are equal to zero, if not, errors are counted. All the tasks of the proposed communication chain are fully vectorized with the MIPP wrapper. The intra-frame SIMD strategy is applied.

This section is decomposed in three sub-sections. The first one details the selected experimentation platforms. Then, the mono-threaded performances are given for each task. Finally, the full potential of the simulator is demonstrated over multi-threaded and multi-node executions.

4.6.1 Experimentation Platforms

Table 4.18 – Specifications of the target processors.

CPU		SIMD instr.		# Proc.	# Cores per Proc.	Freq. (GHz)	SMT	Turbo Boost
		Name	Size					
ARM®	ThunderX2® CN9975	NEON	128-bit	2	28	2.00	4	✗
Intel®	Xeon Phi™ 7230	AVX-512F	512-bit	1	64	1.30	4	✓
Intel®	Xeon™ E5-2680 v3	AVX2	256-bit	2	12	2.50	1	✗
Intel®	Xeon™ Gold 6140	AVX-512BW	512-bit	2	18	2.30	2	✓
Intel®	Xeon™ Gold 6240	AVX-512BW	512-bit	2	18	2.60	1	✗
AMD®	EPYC™ 7702	AVX2	256-bit	2	64	2.00	1	✗

Table 4.18 summarizes the 6 server-class selected CPUs. These CPUs have been chosen because they have a lot of cores and are good candidates for a multi-threading scaling evaluation. All these CPUs are 8-bit SIMD fixed-point capable except for the Xeon Phi™ 7230. In this specific case the computations are made on 32-bit everywhere and the quantizer task is skipped. Also, Intel®, AMD® and ARM® CPUs are representative of the today market. It demonstrates the flexibility and the portability capacities of the proposed simulator. One can note that AFF3CT can adapt to various CPU architectures.

For all the CPU targets, the code has been compiled with the `C++` GNU compiler version 8.2.0 on Linux, with the following optimization flags: `-O3 -funroll-loops -march=native`. Note that AFF3CT also works on Windows and macOS at the same level of performance.

4.6.2 Mono-threaded Performances

Table 4.19 – Average throughput and latency performance per simulated task (single-threaded). In **bold** the best performance by task. In blue the best total simulation performance and in red the worst total simulation performance.

	CPU	Chan.	Demod.	Quant.	Dec.	CRC ext.	Mon.	Total
$\mathcal{T}$ (Mb/s)	ThunderX2® CN9975	53.7	672.9	748.3	112.1	6338.4	2386.3	28.4
	Xeon Phi™ 7230	100.2	1862.1	-	49.8	2073.2	921.4	29.0
	Xeon™ E5-2680 v3	159.7	7427.1	4586.2	247.5	20832.6	8234.5	82.3
	Xeon™ Gold 6140	**421.7**	14131.7	**12931.5**	**376.5**	**31749.5**	11093.0	171.8
	Xeon™ Gold 6240	314.6	10518.3	9915.8	277.4	23118.3	7953.5	127.3
	EPYC™ 7702	215.4	**14919.5**	9234.5	359.5	28404.0	**13562.0**	115.4
$\mathcal{L}$ (µs)	ThunderX2® CN9975	38.15	3.04	2.74	15.65	0.27	0.72	60.58
	Xeon Phi™ 7230	20.45	1.10	-	35.23	0.83	1.87	59.48
	Xeon™ E5-2680 v3	12.82	0.28	0.45	7.09	0.08	0.21	20.93
	Xeon™ Gold 6140	**4.86**	0.14	**0.16**	**4.66**	**0.05**	0.16	10.03
	Xeon™ Gold 6240	6.51	0.19	0.21	6.33	0.07	0.22	13.53
	EPYC™ 7702	9.51	**0.14**	0.22	4.88	0.06	**0.13**	14.94

Table 4.19 presents the mono-threaded throughput ($\mathcal{T}$) and the latency ($\mathcal{L}$) of the simulated tasks. The throughput is calculated depending on the number of output samples in the task. For instance, the throughput of the channel is estimated with N bits while the throughput of the decoder is computed with K_1 bits. The latency is the average latency. The *Total* column corresponds to the global throughput and latency. The total latency is the cumulative latencies of each task while the total throughput is deduced from the total latency and K_2 value.

In general, the most time consuming tasks are the AWGN channel and the decoder. The other tasks are mainly negligible thanks to the fast SIMD implementations. The Intel® Gold 6140 CPU has the best throughputs and latencies in general. This is due to the Turbo Boost that enables the CPU to reach very high frequencies on a single thread but not only. The main difference with the AMD® CPU comes from the best performance of the Gold 6140 CPU for the AWGN channel task. Indeed, the Gold 6140 CPU takes advantage of its doubled SIMD length (AVX-512) compared to the AMD® EPYC CPU (AVX2). The ARM® ThunderX2® CPU comes with the worst single threaded general performance. The Intel® Xeon Phi™ is also very close to the ARM® CPU performance. This is not surprising as these processors have not been designed for the single core performance but for multi-core performance.

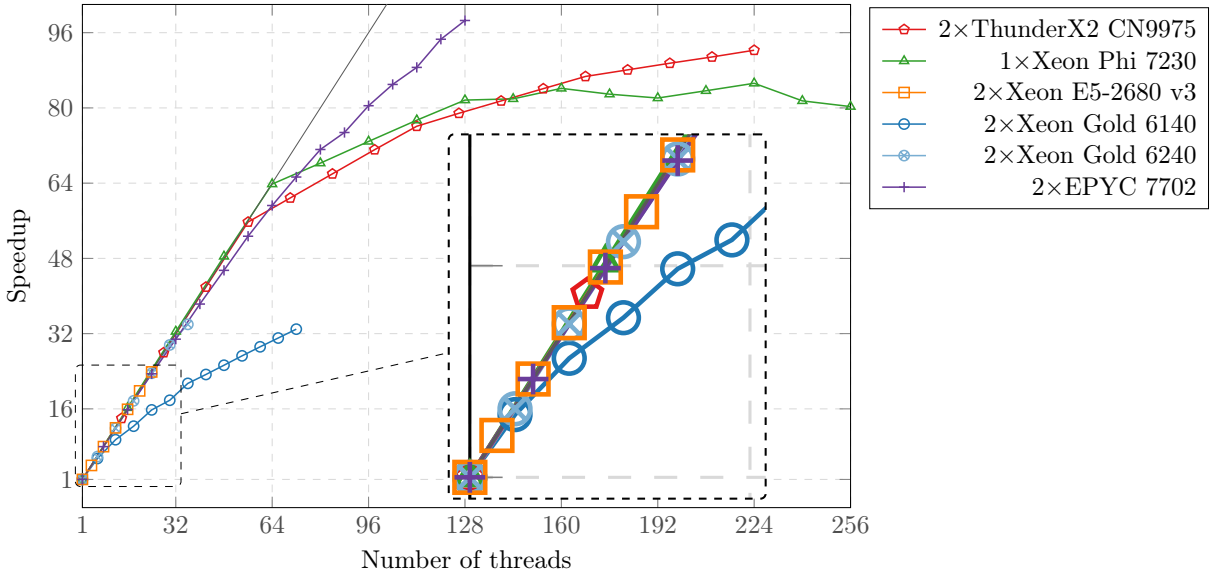
4.6.3 Multi-threaded and Multi-node Performances

Figure 4.13a depicts the speedups achieved on the various modern CPU architectures, while Figure 4.13b exposes the corresponding simulation information throughputs. In Figure 4.13a, the speedups on each architecture are computed with respect to the single thread simulation time on the same architecture. Each run assigns at most one AFF3CT thread to each hardware thread. Since the architectures have different number of hardware threads, the presented speedups do not all have the same number of measurement points. To reduce the simulation time it is possible to multiply the number of concurrent communication chains, thanks to the independence property of Monte Carlo simulations. The simulation scales rather well on the tested architectures. The data remains in the CPU caches because of the moderate frame size ($\mathsf{N} = 2048$) and SIMD intra-frame strategy. Scaling on the Xeon™ Gold 6140 is not as good as the other targets, because the *Intel® Turbo Boost* technology is enabled. The CPU runs at higher frequencies when the number of active cores is low. AFF3CT effectively leverages the simultaneous multi-threading (SMT) technology. This is especially true for the ThunderX2® CN9975 and Xeon™ Gold 6140 targets. The SMT technology helps to improve the usage of the available instruction-level parallelism.

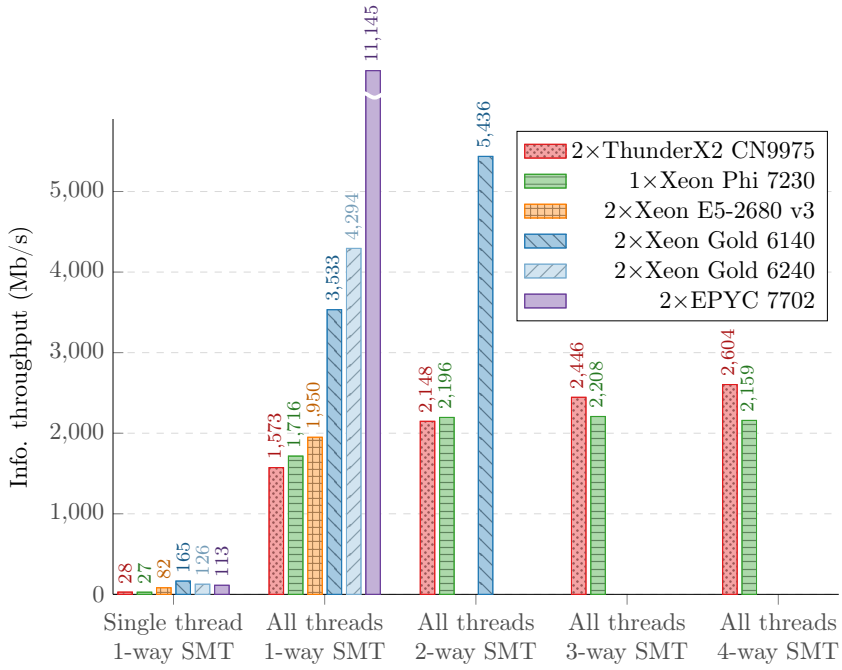
The best multi-threaded throughput performance is achieved on the AMD® EPYC platform and its 128 cores. 11 Gb/s are reached on the two AMD® EPYC 7702 CPUs. The best Intel® target (Xeon™ Gold 6140) can only get half of the AMD® CPU performance. The ARM® target and the Xeon Phi™ are not competitive in terms of throughput even when all the hardware threads are assigned. It is possible that these targets are more adapted for SDR usage (like for the C-RAN) when power consumption matter more. However, for simulation purpose, it is clear that the AMD® EPYC CPUs are more interesting. Those results also demonstrate the interest to enable the frequency boost as well as the SMT technology. Indeed, the Xeon™ Gold 6240 CPU should be faster than the Xeon™ Gold 6140 CPU. But, the frequency boost and the SMT technology have been disabled on this server. It results in worst multi-threaded performance. We could not try to enable the frequency boost and the SMT techniques on the AMD® CPU but there is a good chance that it would have increased the throughput even more.

Table 4.20 shows the multi-node scaling with the OpenMPI library (version 3.1.2). The information throughput ($\mathcal{T}_\mathtt{i}$) and the speedup values are almost linear with the number of nodes: This is expected because there are very few communications between the various MPI processes. Note that the super-linear scaling is due to the measurement imprecision.

Those aforementioned results demonstrate the high throughput capabilities of the AFF3CT simulator. For instance, when using 32 MPI nodes on the given (2048,1723) polar code, it takes about one minute to estimate the $\mathsf{E_b}/\mathsf{N_0} = 4.5$ dB SNR point (BER = 4.34e-10, FER = 5.17e-08).



(a) Simulator speedups.



(b) Simulator throughputs.

Figure 4.13 – AFF3CT simulation results of a (2048,1723) Polar code, FA-SSCL decoder $L = 32$, BPSK modulation, AWGN channel, $E_b/N_0 = 4.5$ dB (BER = 4.34e-10, FER = 5.17e-08).

Table 4.20 – AFF3CT multi-node speedups (single node: 2×Xeon™ E5-2680 v3).

Nodes	Cores	$\mathcal{T}_i$ (Mb/s)	Speedup
1	24	1,950	1.00
2	48	3,901	1.95
4	96	7,793	4.00
8	192	15,829	8.12
16	384	31,640	16.22
32	768	63,075	32.34

4.7 Conclusion

In this chapter we evaluated the implementations of the LDPC decoders, polar decoders and turbo decoders studied during this thesis. The throughput, the latency and the energy efficiency have been studied and compared with other works. For the LDPC decoders and the turbo decoders, the inter-frame strategy has been applied and leads to throughputs comparable with the state-of-the-art performance. However, the latencies are not competitive with the intra-frame implementations found in the literature. These implementations are then more oriented for simulation purpose or for real-time applications that do not require low latency like the video streaming, for instance. For the polar decoders, both the inter-frame and intra-frame strategies has been implemented. It results in a complete framework that can adapt to many applicative contexts. The proposed decoders are ones of the fastest in the literature. There are also able to be very flexible with the dynamic implementations or specialized for highest possible performance with the source code generation technique. For all the proposed decoders (LDPC, polar and turbo), the level of genericity is one of our main contribution. The implementations are able to adapt to various CPUs architectures as well as to support many algorithmic variants. Moreover, each of the presented implementations are able to work close to the reference decoding performance. Most of the obtained results have been published in scientific conferences and journals [2, 3, 4, 5, 6].

The software decoder Hall of Fames (HoFs) are then introduced. These HoFs are exhaustive surveys of the software decoders found in the literature. The proposed decoders are reported as well as the other state-of-the-art works. These HoFs enables to compare CPU and GPU implementations. Some metrics like the normalized throughput, the TNDC and the energy consumption are defined. The results show that these last years, the CPU implementations are more efficient than the GPU works in terms of throughput, latency and energy efficiency. One of the main issue of the GPU-based implementations is the required transfer time between the CPU and the GPU. An other main issue comes from the intrinsic architecture of the GPU that requires a lot of parallelism to be efficient. This is not always possible to take advantage of this high level of parallelism in the channel decoding algorithms. As a consequence, in general, the CPUs are more adapted to low latency implementations than the GPUs.

The last section of this chapter focuses on the AFF3CT simulator performance. A fully vectorized digital communication chain is proposed for the evaluation. First the mono-threaded performances are reported. As a result, AFF3CT runs fast on the last Intel® Gold CPUs that support the AVX-512 SIMD engine. Then the multi-threaded performances are benched and the AMD® EPYC CPUs comes with the best throughout performance, up to 11 Gb/s. Even if the AMD® EPYC CPUs only supports the AVX instructions, it looks like the Zen 2 architecture is well balanced between computational power and memory speed. Finally, the multi-node capacity of the AFF3CT simulator is tested and a linear speedup is observed over 32 nodes. The peak throughput performance in multi-node is 32 Gb/s. These high throughputs enable the exploration of many combinations at very low error-rate level. Preliminary results have been published in a scientific journal [1]. To the best of our knowledge, AFF3CT is one of the current fastest FEC simulator.

The next chapter presents a new extension of the AFF3CT library to improve the SDR support. An embedded domain specific language is proposed to ease the usage of multi-core CPUs in real-time contexts.

5 Embedded Domain Specific Language for the Software-defined Radio

This chapter presents a new embedded Domain Specific Language (eDSL) dedicated to the Software-Defined Radio (SDR). The first section discusses the existing models and solutions. It also motivates the need of a new dedicated language for the SDR. In a second part a description of the proposed eDSL is given and detailed in two sub-sections: the elementary components are presented first and then the parallel components are described. The third part is focusing on the implementations of the previously presented components. Among others, the sequence duplication technique and the pipeline implementation are discussed. Finally, the last part shows a concrete use case of the proposed eDSL on the well-spread DVB-S2 standard. A fully digital transceiver has been designed in software. The DVB-S2 standard is presented from an applicative point of view (transmitter and receiver) and is then evaluated on a specific CPU target.

5.1 Related Works

Digital communication systems are traditionally implemented onto dedicated hardware (ASIC) targeting high throughputs, low latencies and energy efficiency. However, the implementations of such solutions suffer from a long time to market, are expensive and are specific by nature [Pal+10, Pal+12].

The new standards like the 5G are coming with very large specifications and multiple possible configurations [ETS18]. Small objects that need to communicate very few data at low rates are going to live together with 4K video streaming for mobile phone games which will require high throughputs as well as low latencies [Ros+14].

To meet those various specifications the transceivers have to be able to adapt quickly to new configurations. There is a growing need for flexible, re-configurable and programmable solutions. To match those constraints, there is a growing interest for the SDR which consists in processing both the Physical (PHY) and Medium Access Control (MAC) layers in software [Mit93], as opposed to the traditionally hardware-based solutions. Short time to market, lower development costs, interoperability, readiness to adapt to future updates and new protocols are the mains advantages of the SDR [AD18].

The SDR can be implemented on various targets like Field Programmable Gate Arrays (FPGAs) [CC04, SBW06, Dut+10, SA13, MBB15, NRV16], Digital Signal Processors (DSPs) [KR08, Kar+13, SA13] or General Purpose Processors (GPPs) [YC12, Ban+14, MK19, GU20]. Many elementary blocks of digital communication systems have been optimized to run fast on Intel® and ARM® CPUs [4, 6, 5, 7, 3, 2] and have been packaged in AFF3CT [8, 1]. Even if there is some interesting results in term of throughput on GPUs [Xia+13, Li+14, LJC14, Gia+16, KK17a], the achieved latency on this architecture is still too high to meet real time constraints and cannot compete with existing CPU implementations [LLJ15, 4, Gia+16, 6, LJ17, 3, LJ19]. This is mainly due to data transfers between the host (CPUs) and the device (GPUs), and to the intrinsic nature of the GPU architecture which is not optimized for latency efficiency.

5.1.1 Dataflow Model

The Shanon's communication model presented in Section 1.1 can be refined in a way that the transmitter and the receiver are decomposed into many processing blocks. Those blocks are mainly connected to each other in a directed graph. This perfectly matches the dataflow model [Den80, Ack82]: the blocks are the filters and the binding links between the blocks represent the data exchanges. The dataflow model enables to describe the system from a high level point of view and to perform optimizations independently of the system designer. In the case of the SDR, simpler models than the generalized dataflow can be used like the synchronous dataflow [LM87] or the cyclo-static dataflow [Eng+94, Bil+95]. It enables to perform aggressive simplifications like a static scheduling of the filters execution [PPL95].

5.1.2 Dedicated Languages

Many languages dedicated to streaming applications have been introduced [Buc+04, Ama+05, Lia+06, BD10, GDB10, TA10, DLB17]. Streaming applications are most of the time represented with the dataflow model and the dedicated languages often support the general or the cyclo-static dataflow model. They also very often come with automatic parallelism mechanisms like pipelining

and forks/joins. With dedicated languages, it is natural to describe a system with the dataflow model. The main drawbacks of this type of solution is that the designer has to learn a new language. But, when something is not intended by the language it is hard to find a workaround.

5.1.3 GNU Radio

There are few solutions targeting specifically the SDR sub-domain yet. The most famous is GNU Radio [Ron+06] which is open source and largely adopted by the community. The software is bundled with a large variety of algorithms used in real life systems. GNU Radio models digital communication systems at the symbol level. This philosophy is very close to the algorithms descriptions that can be found in the signal community literature. Thus, it enables quick implementation of new digital processing algorithms. Still, this is a limitation to meet high throughputs and low latencies constraints on current GPPs architectures.

5.2 Description of the Proposed Embedded Domain Specific Language

This section details, through AFF3CT, our proposition of an embedded Domain Specific Language (eDSL) working on sets of symbols (aka frames). AFF3CT implements a form of the synchronous dataflow (SDF) model dedicated to the relevant characteristics of FEC digital communication chains. It performs more aggressive optimizations than GNU Radio, at the cost of lower generality.

Many `C++` eDSLs are based on the template meta-programming technique. It performs computations at the compilation time instead of during the program execution. In AFF3CT we decided to not use the `C++` template meta-programming for the eDSL. This choice has been made to match the existing internal organization of the project and more precisely the *modules*, *tasks* and *sockets* components. It was also comforted by one of the main draw back of the template-based eDSLs: the errors management and reporting. A non-negligible part on the proposed eDSL is dedicated to error verifications and clear messages display. Of course the static scheduling implied by the synchronous dataflow model could theoretically be resolved at the compilation time. Instead of that, the scheduling of the tasks is precomputed at the runtime and the execution order is stored in an array of function pointers (`std::function`). The cost of the execution of the tasks is very cheap (consecutive function calls). It means that the overhead is negligible compared to a template-based solution.

5.2.1 Elementary Components

The proposed eDSL comes with a set of elementary components: *sequence*, *module*, *task* and *socket*. The *module*, *task* and *socket* components have been reused and enriched from the AFF3CT library (see Section 3.3.2).

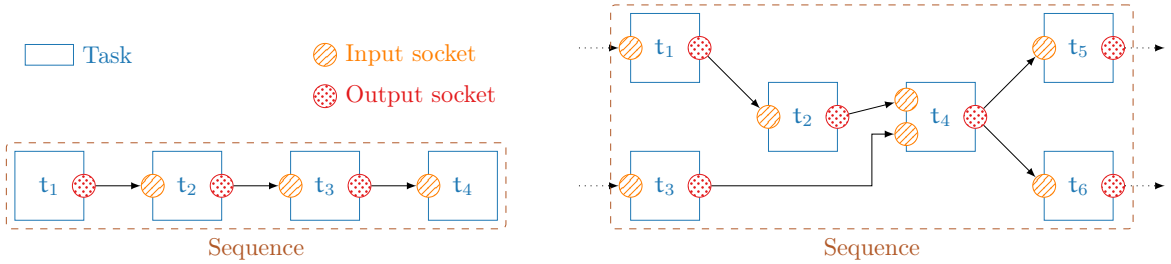
In the eDSL the *task* is the fundamental component. It is also known as the *filter* in the standard dataflow model. A task is an elementary component, it can be an encoder, a decoder or a modulator processing for instance. A task, unlike a dataflow filter, can have an internal state and a private memory to store temporary data. If the lifetime of the private data exceeds the task execution then the data is owned by the *module*. Additionally a set of tasks can share the

same internal/private memory, in that case, multiple tasks are regrouped in a single module. This behavior is not recommended by the standard dataflow model. It should be avoided in a fully dataflow-compliant model. The main problem with internal memory is that the tasks cannot be executed safely by many threads in parallel because of data races. However, in many situations the writing of a task or a set of tasks can be simplified by relieving this constraint. Furthermore, storing private data in the module can be, in some cases, a way to avoid to allocate memory for each task execution which is expensive.

A task can consume and produce public data. To this purpose, each task exposes input and/or output *sockets*. The action of connecting the sockets of different tasks is called the *binding* (see Figure 3.1). The binding determines the tasks execution order.

Tasks that have been bound together can be regrouped in what we call a *sequence* of tasks. The sequence notion reminds us that the tasks are executed sequentially in a fixed order (like in the SDF model). To create a sequence, the designer has to give the first tasks and the last tasks to execute. Then the connected tasks will be analyzed and a sequence object will be built. The analysis is a deep traversal of the tasks graph. The order in which the tasks have been traversed is memorized in the sequence. When the designer calls the `exec` method on a sequence, the tasks constituting it are executed one by one in the memorized order (static scheduling).

The proposed eDSL is targeting streaming applications and more precisely signal processing and digital communication chains. In this type of applications, the processing is repeated indefinitely on batches of frames when the system is on. So, a sequence is executed in loop, it means that when the last task is executed, the next task is the first one on the next frame. The designer can control if the sequence should restart by giving a *condition function* when calling the `exec` method on the sequence. The prototype of the function is `bool cond_func(void)`: if the function returns *false*, the sequence is repeated, the sequence is stopped otherwise.



(a) Simple chain sequence.

(b) Sequence with multiple first and last tasks.

Figure 5.1 – Example of sequences.

Figure 5.3 shows two examples of sequence. Figure 5.1a is a simple chain of tasks. The designer only needs to specify the first task of the chain (t_1). The sequence analysis automatically follows the binding until the last task (t_4). Figure 5.1b is more complicated. Indeed there are bound tasks before and after the wanted sequence. There are also two *first* tasks (t_1 and t_3) and two *last* tasks (t_5 and t_6). In this case, the designer needs to explicitly specify that t_1 and t_3 are first tasks. If t_1 is given before t_3 then t_1 will be executed first and t_3 after. The analysis starts from t_1 and continue to traverse new tasks if possible. In this example, t_2 can be executed directly after t_1 but t_4 cannot because it depends on t_3. So the analysis stops after t_2 and then restarts from t_3. The index i of the t_i task represents the execution order. The t_5 and t_6 last tasks have to be given by the designer because there are other tasks bound to their output sockets: the analysis cannot guess the end of the sequence alone.

In general, all the tasks of a sequence are repeated in the same predefined order. But it some particular cases, a task can raise the `tools::processing_aborted` exception to restart the sequence (on the next frame) before its end. For instance, in Figure 5.1a if the t_3 task raises the `tools::processing_aborted` exception, then the next executed task is t_1 instead of t_4.

Some digital communication scenarios include repeated schemes. To map a repeated scheme, we introduced a specific type of task: the *loop*. This task is executed one or more times depending on a *condition*. A loop is bound to two sub-sequences of tasks. The first one is executed and repeated while the condition is *false* and the second one is taken when the condition is *true*.

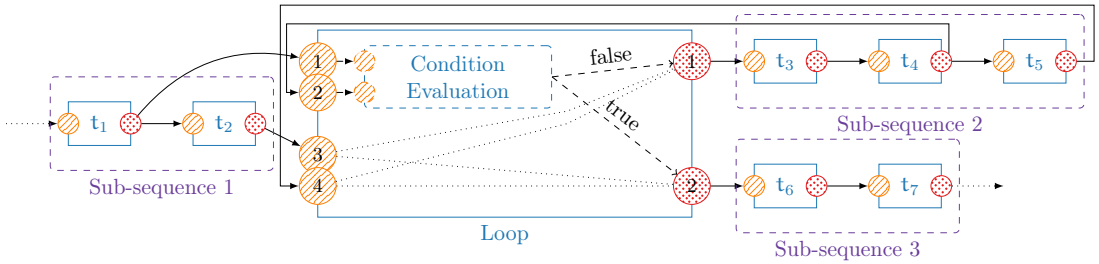


Figure 5.2 – Example of a sequence of tasks with a loop.

Figure 5.2 illustrates a sequence of tasks with a loop. In this particular case, the loop condition is based on an input socket. So the condition evaluation is dynamic and depends on the runtime values of this input socket. In a first place, the sub-sequence 1 is executed; then the loop condition is evaluated; if the condition return *false* the sub-sequence 2 is executed and the loop condition is re-evaluated until it returns *true*. At this point the sub-sequence 3 is executed. After the execution on the sub-sequence 1, the convention in the loop is to use the input sockets 1 and 3. The input socket 1 is only applied for the condition evaluation whereas the input socket 3 is simply forwarded to the output socket 1 and 2. If the sub-sequence 2 is executed, the input sockets 2 and 4 will be used until the end of the loop.

In the example the loop uses an input socket to evaluate the condition. This is common in iterative demodulation/decoding schemes when the overall system can have an early termination criterion like a CRC detection. Of course, it is also possible to override the loop behavior by inheriting from it. The condition evaluation process can be modified by the designer. The loop can be a predicate: in this case, the condition evaluation does not require any input socket. The predicate can be a counter. Each time the condition is evaluated, the counter is incremented. When the counter reaches a given value then the condition evaluation returns *true* (*for-loop* behavior).

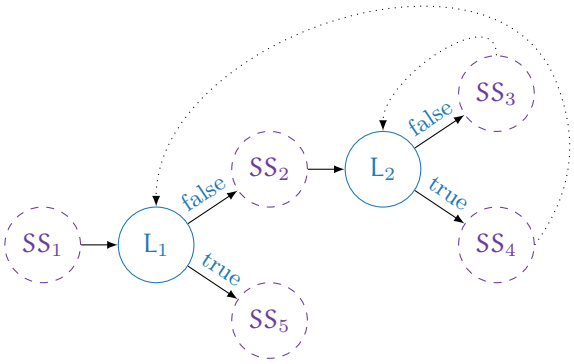


Figure 5.3 – Nested loops.

The overall system also supports nested loops. The idea is to regroup the tasks in sub-sequences: one before the loop and two after the loop. This is implemented as a binary tree. Each time a loop is encountered, two new paths are created. Figure 5.3 shows an example with two loops (L_1 and L_2). Five sub-sequences of tasks are created (SS_x): one before L_1 (SS_1), two after L_1 (SS_2 and SS_5) and two after L_2 (SS_3 and SS_4). In this example L_2 is a nested loop. Indeed, it is an inner loop within the body of an outer one (L_1).

5.2.2 Parallel Components

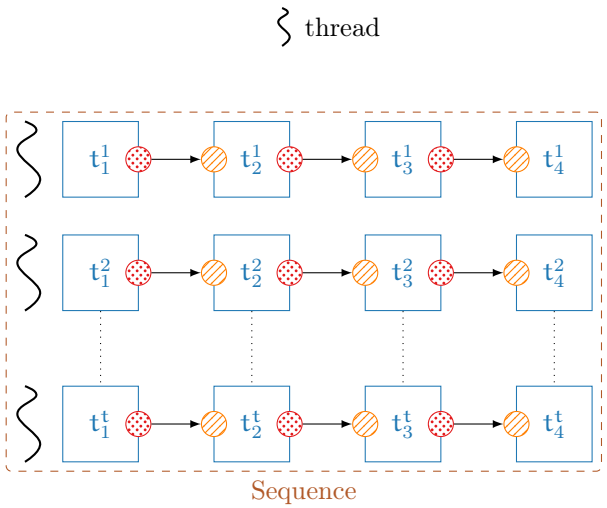


Figure 5.4 – Sequence duplication for multi-threaded execution.

A created sequence can be duplicated. This way many threads can execute the sequence in parallel. The `t` number of duplications (= number of threads) is a parameter of its constructor. As shown in Figure 5.4, one thread is affected to one duplicated chain. Thus, a sequence is able to take advantage of the multi-core architectures. For instance, the AFF3CT simulator, extensively exploits the sequence duplication feature to automatically parallelize the processing. The duplication strategy is efficient since no synchronization is necessary between the threads. Each threaded sequence can be executed on one dedicated core and the public data transfers remain on this core for the data reuse in the caches. However this parallelism is only possible if the tasks themselves can be duplicated.

In some particular cases like in the signal synchronization processing, the tasks can have a dependency on themselves. It is then impossible to duplicate the sequence because of the sequential nature of the tasks. To overcome this issue the well-known pipelining strategy can be applied to increase the sequence throughput up to the slowest task throughput. The proposed eDSL comes with a specific *pipeline* component to this purpose. The pipeline takes multiple sequences as input. Each sequence of the pipeline is called a *stage*. A pipeline stage is run on one thread. For instance, a 4-stage pipeline creates 4 threads, one thread per stage. A pipeline stage can be combined with the sequence duplication strategy. It means that there are nested threads in the current stage thread. Be aware that the pipelining strategy comes with an extra synchronization cost between the stage threads. The implementation details will be discussed in the next section. It worth mentioning that is not possible to split a loop in separated pipeline stages.

5.3 Implementation Strategies

5.3.1 Implicit Rules

In the proposed eDSL, an input socket can only be bound to one output socket while an output socket can be bound to multiple input sockets. A task can only be executed if all its input sockets are bound. In a sequence, the scheduling of the tasks is defined by the binding order. The general rule in the sequence analysis is to add a task to the array of function pointers when its last input socket is discovered in the deep first traversal of the tasks graph. After that, the output sockets of the current task are followed to reach new tasks. The new tasks are discovered in the order in which they were bound by the designer.

5.3.2 Sequence Duplication

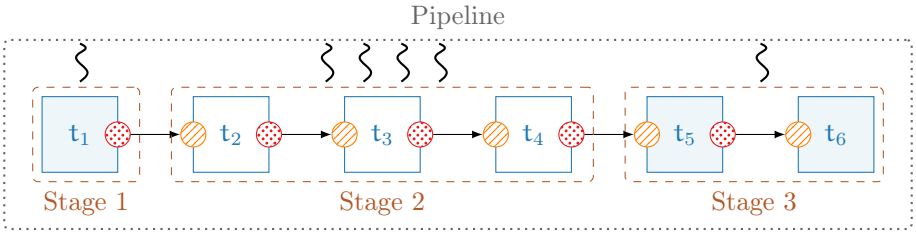
In order to duplicate sequences, a *clone* method is implemented in the modules. The *clone* method is polymorphic and defined in the *Module* abstract class. It relies on the implicit copy constructors and a *deep copy* protected method (overridable). The *clone* method prototype is `module::Module* clone() const`. In an implementation (*ModuleImpl*) of the abstract *Module* class, a covariant return type is used: `module::ModuleImpl* clone() const`. The *clone* method implementation first calls the implicit copy constructor of the *ModuleImpl* class and secondly calls the *deep copy* protected method. It is the responsibility of the *ModuleImpl* developer to correctly override the *deep copy* method.

In a fully dataflow-compliant model, there is no need to duplicate the sequence because a filter (or task) is always thread-safe. In the proposed eDSL, we had to introduce the *clone* method because a task can have an internal state and use private memory (stored in the module). It means that this task is not thread-safe. The *deep copy* method deals with pointer and reference members. If the pointer/reference members are read-only (`const`), then the implicit copy constructor copies the memory addresses automatically. The problem comes when there is writable pointer/reference members. If the current *ModuleImpl* class possesses one ore more writable references then it means that the module can't be cloned. The tasks of the module are sequential. In the particular case of a writable pointer member, the developer can explicitly allocate a new pointer in the *deep copy* method. Note that if a task does not implement the *clone* method, the eDSL outputs an error message during the sequence analysis if the designer tries to make a duplication.

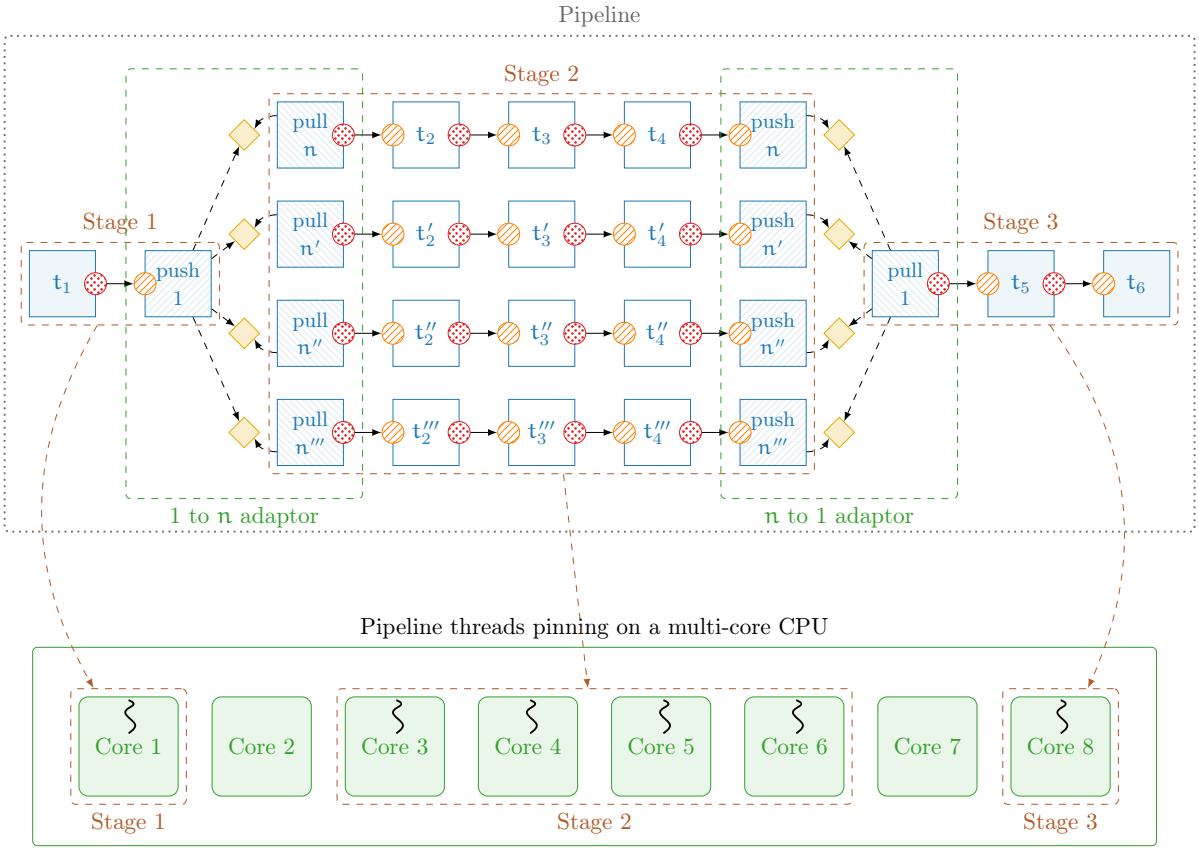
5.3.3 Processes

A sequence encapsulates a set of tasks and gives the opportunity to execute these tasks in a predefined order. The designer can also execute tasks explicitly, outside sequences. With a sequence, the analysis is able to match specific patterns (known configurations of bound tasks) and replace them by a more efficient source code. To this purpose the notion of *process* has been introduced: in a sequence each task is encapsulated in a *process* (this has nothing to do with OS processes). For the majority of the tasks, the *process* just executes the task. But for some specific patterns, the task execution source code is replaced by a more efficient one. The pattern detection is based on a `C++` introspection feature: during the analysis, for each parsed task, we try to cast (`dynamic_cast`) the corresponding module in a specific class. In the next section, the processes are applied to improve the efficiency of the pipeline.

5.3.4 Pipeline



(a) Description of a pipeline: tasks creation, tasks binding and sequences/stages definition (with the corresponding number of threads).



(b) Automatic parallelization of a pipeline description: sequence duplications, 1 to n and n to 1 adaptors creation and binding.

Figure 5.5 – Example of a pipeline description and the associate transformation with adaptors.

In this section, the pipeline implementation is illustrated through a simple example. Figure 5.5 shows the difference between a pipeline description (see Figure 5.5a) and its actual instantiation (see Figure 5.5b). In Figure 5.5 we suppose that the t_1, t_5 and t_6 tasks cannot be duplicated (filled in blue). The designer knows that the execution time of the t_1 task is higher than the cumulate execution time of t_5 and t_6 tasks. We assume that the cumulate execution time of t_2, t_3 and t_4 is approximatively four times higher than t_1. This knowledge motivates the splitting of the stages 1, 2 and 3. There is no need to split the t_5 and t_6 tasks in two stages because the overall throughput is limited by the slowest stage (t_1 here). The stage 2 is duplicated four times to increase its throughput by four as we know that its latency is approximatively four times the sequence 1. In general, a primarily profiling phase of the sequential code is required to define the pipeline strategy. Listing 5.1 presents the `C++` eDSL source code corresponding to the pipeline description in Figure 5.5a. Each t_i task is contained (as a method) in the M_i module (or class).

```cpp
#include <aff3ct.hpp>
using namespace aff3ct;

int main()
{
    // 1) creation of the module objects
    module::M1 m1_obj(/* ... */); // 'M1' class contains 't1' task
    module::M2 m2_obj(/* ... */); // 'M2' class contains 't2' task
    module::M3 m3_obj(/* ... */); // 'M3' class contains 't3' task
    module::M4 m4_obj(/* ... */); // 'M4' class contains 't4' task
    module::M5 m5_obj(/* ... */); // 'M5' class contains 't5' task
    module::M6 m6_obj(/* ... */); // 'M6' class contains 't6' task

    // 2) binding of the tasks
    m2_obj[module::m2::sck::t2::in].bind(m1_obj[module::m1::sck::t1::out]);
    m3_obj[module::m3::sck::t3::in].bind(m2_obj[module::m2::sck::t2::out]);
    m4_obj[module::m4::sck::t4::in].bind(m3_obj[module::m3::sck::t3::out]);
    m5_obj[module::m5::sck::t5::in].bind(m4_obj[module::m4::sck::t4::out]);
    m6_obj[module::m6::sck::t6::in].bind(m5_obj[module::m5::sck::t5::out]);

    // 3) creation of the pipeline (= sequences and pipeline analyses)
    tools::Pipeline pipeline(
        // first task of the sequence (for validation purpose)
        m1_obj[module::m1::tsk::t1],
        // description of the sequence decomposition in stages
        { // pipeline stage 1
          { { m1_obj[module::m1::tsk::t1] },   // first task of stage 1
            { m1_obj[module::m1::tsk::t1] } }, // last  task of stage 1
          // pipeline stage 2
          { { m2_obj[module::m2::tsk::t2] },   // first task of stage 2
            { m4_obj[module::m4::tsk::t4] } }, // last  task of stage 2
          // pipeline stage 3
          { { m5_obj[module::m5::tsk::t5] },   // first task of stage 3
            { m6_obj[module::m6::tsk::t6] } }, // last  task of stage 3
        },
        // number of threads per stage (4 sequence duplications in stage 2)
        { 1, 4, 1 }, /* ... */
        // explicit pinning of the threads
        {
          {          1 }, // pin thread  '1'       of stage 1 to core  '1'
          { 3, 4, 5, 6 }, // pin threads '1,2,3,4' of stage 2 to cores '3,4,5,6'
          {          8 }, // pin thread  '1'       of stage 3 to core  '8'
        });

    // 4) execution of the pipeline, it is indefinitely executed in loop
    pipeline.exec([]() { return false; });

    return 0;
}
```

Listing 5.1 – AFF3CT C++ eDSL source code of the pipeline described in Figure 5.5.

The four main steps are: 1) the creation of the modules; 2) the binding of the tasks; 3) the creation of the pipeline; 4) the pipeline execution.

Figure 5.5b presents the internal structure of the pipeline. As we can see, new tasks have been automatically added: *push 1*, *pull n* shared by a *1 to n adaptor* module and *push n*, *pull 1* shared by a *n to 1 adaptor* module. The binding as been modified to insert the tasks of the adaptors. For instance, in the initial pipeline description, $\mathtt{t}_1$ is bound to $\mathtt{t}_2$. In a parallel pipelined execution this is not possible anymore because many threads are running concurrently: one for $\mathtt{t}_1$, four for $\mathtt{t}_2$ and one for $\mathtt{t}_3$ in the example. To this purpose, the adaptors implement a producer-consumer algorithm. The yellow diamonds represent the buffers of the producer-consumer algorithm. The *push 1* and *pull 1* tasks can only be executed by a unique thread while the *pull n* and *push n* tasks support to be executed by multiple threads concurrently. The *push 1* task copies its input socket in one buffer each time it is called. There is one buffer per duplicated sequence (or thread). To guarantee that the order of the input frames is conserved, a round-robin scheduling has been adopted (at the first call, a copy to the first buffer is performed, at the second call, a copy to the second buffer is performed and so on). On the other side, the *pull 1* task is copying the data from the buffers to its output socket, with the same round-robin scheduling.

During the pipeline creation, it is possible to select the size of the synchronization buffers in the adaptors. The default buffer size is one (the buffers can only contain the data of one *push 1* input sockets). During the copy of the input sockets data in one of the buffers, the corresponding thread cannot access the data until the copy is finished. The synchronization is automatically managed by the framework. If the buffer is full, the producer (*push 1 and n* tasks) has to wait. It is similar for the consumer (*pull 1 and n* tasks) if the buffer is empty. We implemented both active and passing waiting.

In the previously described implementation, the copies from and to the buffers can take a non-negligible amount of time. Thus, the process encapsulation detailed before replaces the copies by pointer switches. The idea is to dynamically re-bind the tasks just before and just after the *push* and *pull* tasks. It is also necessary to bypass the regular execution in the *push 1*, *pull n*, *push n* and *pull 1* tasks. The processes that encapsulate these tasks dynamically replace the source code of the data buffer copy by a simple pointer copy. The pointers are exchanged cyclically.

In Figure 5.5b, the pipeline threads are pinned to specific CPU cores. This is the direct consequence of the lines 38-43 in Listing 5.1. The *hwloc* library [Bro+10] has been used and integrated in AFF3CT to pin the software threads to processing units (PUs or hardware threads). In the given example, we assume that the CPU cores can only execute one hardware thread (SMT off) and so an *hwloc* PU is equal to a CPU core. The threads pinning is given by the designer. It can improve the multi-threading performance when dealing with NUMA architectures.

Figure 5.5 is an example of a simple chain of tasks. More complicated task graphs can have more than two tasks to synchronize between two pipeline stages. The adaptor implementation can manage multiple sockets synchronization. The key idea is to deal with a 2-dimensional array of buffers. Another difficult case is when a task $\mathtt{t}_1$ is in stage 1 and possesses an output socket bound to an other task $\mathtt{t}_\chi$ which is located in the stage 4. To work, the pipeline adaptors between the stages 1 and 2 and the stages 2 and 3 automatically synchronize the data of the $\mathtt{t}_1$ output socket.

5.4 Application on the DVB-S2 Standard

The second generation of Digital Video Broadcasting standard for Satellite (DVB-S2) [ETS05] is a flexible standard designed for broadcast applications. DVB-S2 is typically used for the digital television (HDTV with H.264 source coding). In this section, a concrete use case of the AFF3CT eDSL is detailed. The full DVB-S2 transmitter and receiver are implemented in a SDR-compliant system. Two Universal Software Radio Peripherals (USRPs) N320 [1] have been used for the analog signal transmission and reception where all the digital processing of the system have been implemented under AFF3CT. The purpose of this section is not to detail precisely all the implemented tasks but to expose the system as a whole. Some specific focuses are given to describe the main encountered problems and the adopted solutions.

5.4.1 Transmitter Software Implementation

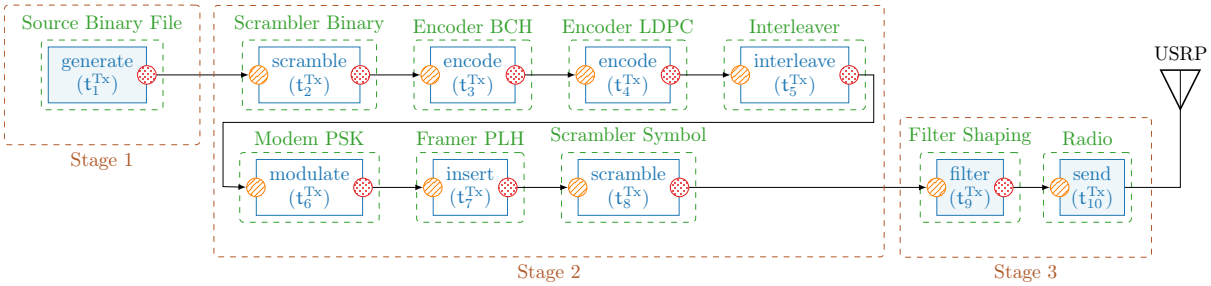


Figure 5.6 – DVB-S2 transmitter software implementation.

Figure 5.6 shows the DVB-S2 transmitter decomposition in tasks. The tasks filled in blue are intrinsically sequential and cannot be duplicated. The initial information bits are read from a binary file (t_1^{Tx}). Then, the DVB-S2 coding scheme rests upon the serial concatenation of a BCH (t_3^{Tx}) and an LDPC code (t_4^{Tx}). The selected modulation (t_6^{Tx}) is a Phase-Shift Keying (PSK). The scrambler tasks (t_2^{Tx} and t_8^{Tx}) apply predefined repeated sequences of *xor* to the frame in order to avoid too long sequences of the same bit or symbol in the frames sent by the radio (t_{10}^{Tx}). Depending on the DVB-S2 configuration (MODCOD), the frame can be interleaved (t_5^{Tx}) after the encoding process. If there is no interleaver, then the frame is just copied. After the modulation, Pay Load Header (PLH) and pilots are inserted (t_7^{Tx}). These extra data are used by the synchronization tasks in the receiver side. Before the radio transmission (t_{10}^{Tx}), the signal bandwidth is rescaled by a shaping filter (t_9^{Tx}).

Table 5.1 – Selected DVB-S2 configurations (MODCOD).

Config.	Modulation	Rate R	K_{BCH}	K_{LDPC}	N_{LDPC}	N_{PLH}	Interleaver
MODCOD 1	QPSK	3/5	9552	9720	16200	16740	no
MODCOD 2	QPSK	8/9	14232	14400	16200	16740	no
MODCOD 3	8-PSK	8/9	14232	14400	16200	16740	column/row

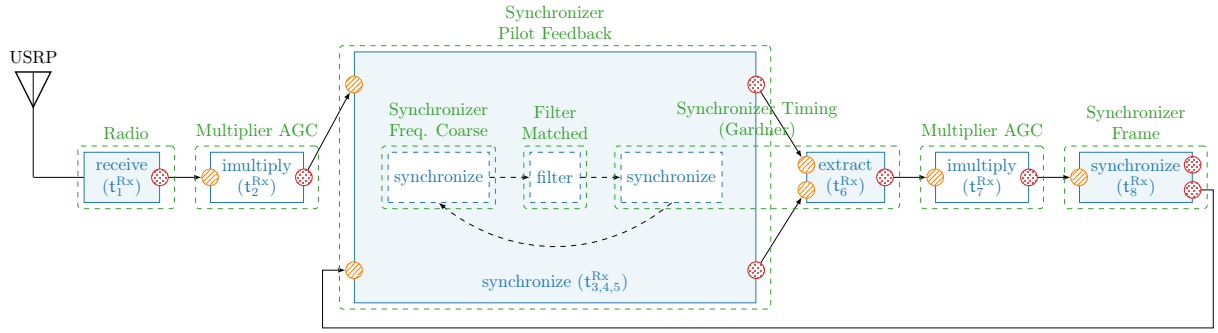
The DVB-S2 defines 32 different configurations or *MODCODs*. This work focuses on the 3 MODCODs given in Table. 5.1. Depending on the MODCOD, the PSK modulation and the LDPC code rate R vary. In the MODCOD 1 and 2 there is no interleaver and the MODCOD 3 uses a column/row interleaver. K_{BCH} or K is the number of information bits and the input size of the BCH encoder. N_{BCH} or K_{LDPC} is the output size of the BCH encoder and the input size of

1. USRP N320: https://www.ettus.com/all-products/usrp-n320/.

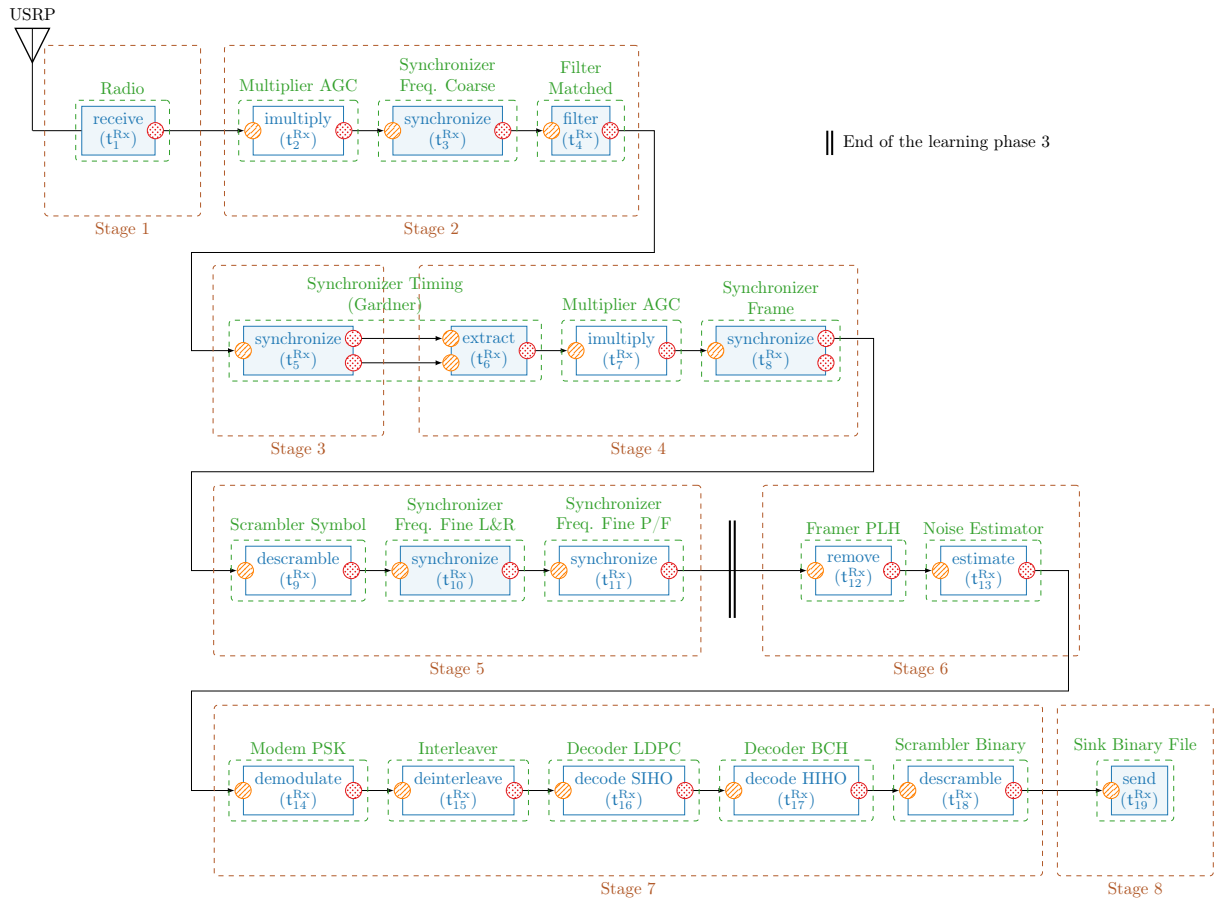
the LDPC encoder. N_{LDPC} is the output size of the LDPC encoder. N_{PLH} or N is the frame size containing N_{LDPC} bits plus the PLH and pilots bits.

The DVB-S2 transmitter software implementation has been split into 3 pipeline stages, the stages 1 and 3 are sequential and the stage 2 is parallel. The transmitter is not the most resources consuming part. An Intel® Core™ i7 CPU with 4 cores (SMT was switched on) has been used. One core has been assigned for the radio thread (stage 3), one hardware thread for the *source* (stage 1) and the five remaining hardware threads have been dedicated to the stage 2.

5.4.2 Receiver Software Implementation



(a) Waiting phase and learning phase 1 & 2.



(b) Learning phase 3 & transmission phase.

Figure 5.7 – DVB-S2 receiver software implementation.

Figure 5.7 presents the task decomposition of the DVB-S2 receiver software implementation with the five distinct phases. The first one is called the *waiting phase* (see Figure 5.7a). It consists in waiting until a transmitter starts to transmit. The *Synchronizer Frame* task (t_8^{Rx}) possesses a frame detection criterion. When a signal is detected, the *learning phase 1* (see Figure 5.7a) is executed during 150 frames. After that the *learning phase 2* (see Figure 5.7a) is also executed during 150 frames. After the *learning phase 1 and 2*, the tasks have to be re-bound for the *learning phase 3* (see Figure 5.7b). This last learning phase is applied over 200 frames. After 500 frames for all the learning phases, the final *transmission phase* is established (see Fig 5.7b).

In a real life communication systems, the internal clocks of the radios can diverge slightly. A specific processing has to be added in order to be resilient. This is achieved by the *Synchronizer Timing* tasks (t_5^{Rx} and t_6^{Rx}). Similarly, the radio transmitter frequency is not exactly the same as the receiver frequency, the *Synchronizer Frequency* tasks (t_3^{Rx}, t_{10}^{Rx} and t_{11}^{Rx}) recalibrate the signal to recover the transmitted symbols. Finally the LDPC is a block coding scheme that requires to know precisely the first and last bits of the codeword. The *Synchronizer Frame* task (t_8^{Rxt}) uses the *Pay Load Headers* (PLH) and pilots bits inserted by the transmitter *Framer PLH* task (t_8^{Tx}) to recover the first and last symbols. One can notice that the *Synchronizer Timing* module is composed by two separated tasks (*synchronize* or t_5^{Rx} and *extract* or t_6^{Rx}). This behavior is different from the other *Synchronizer* modules. The *synchronize* task (t_5^{Rx} or $t_{3,4,5}^{Rx}$) has two output sockets, one for the regular data and another one for a mask. The regular data and the mask are then used by the *extract* task (t_5^{Rx}) to filter which data that is selected or not for the next task. This specific implementation has been retained for two reasons: 1) the *Synchronizer Timing* tasks (t_5^{Rx} and t_6^{Rx}) have a high latency compared to the others tasks and splitting the treatment in two tasks is a way to increase the throughput of the pipeline (this will be discussed more precisely after) and 2) the *extract* task (t_6^{Rx}) introduces a new possible behavior: in some case the task does not have enough samples to produce a frame. In this particular case, the *extract* task raises the `tools::processing_aborted` exception. The exception is caught and the sequence restarts from the first task (t_1^{Rx}). This implies to manage a buffer of samples in the *extract* task (t_6^{Rx}). If the buffer contains more than one frame then the next task (t_7^{Rx}) can be executed, else the sequence has to be restarted.

During the waiting and learning phases 1 and 2, the *Synchronizer Freq. Coarse*, the *Filter Matched* and a part of the *Synchronizer Timing* have to work symbol by symbol. They have been grouped in the *Synchronizer Pilot Feedback* task ($t_{3,4,5}^{Rx}$). $t_{3,4,5}^{Rx}$ also requires a feedback input from the *Synchronizer Frame* task (t_8^{Rx}). This behavior is no longer required in the next phases and so the $t_{3,4,5}^{Rx}$ task has been split in t_3^{Rx}, t_4^{Rx} and t_5^{Rx}. Moreover, the feedback from the t_8^{Rx} second output socket is left unbound.

Figure 5.8 contains the BER and FER decoding performance of the 3 selected MODCODs. The forms represent the channel conditions: the squares stand for a standard simulated AWGN channel, the triangles are also a simulated AWGN channel in which frequency shift, phase shift and symbol delay have been taken into account and the circles are the real conditions measured performances with the USRPs. One can notice a 0.2 dB inaccuracy in the noise estimated by the t_{13}^{Rx} task. It is symbolized by the extra horizontal bars over the circles. The MODCOD 1 is represented by dashed lines; the MODCOD 2 is represented by dotted lines; the MODCOD 3 is represented by solid lines. For each MODCOD, the LDPC decoder is based on the belief propagation algorithm with horizontal layered scheduling (10 iterations) and with the min-sum node update rules. Each DVB-S2 configuration has a well-separated SNR predilection zone.

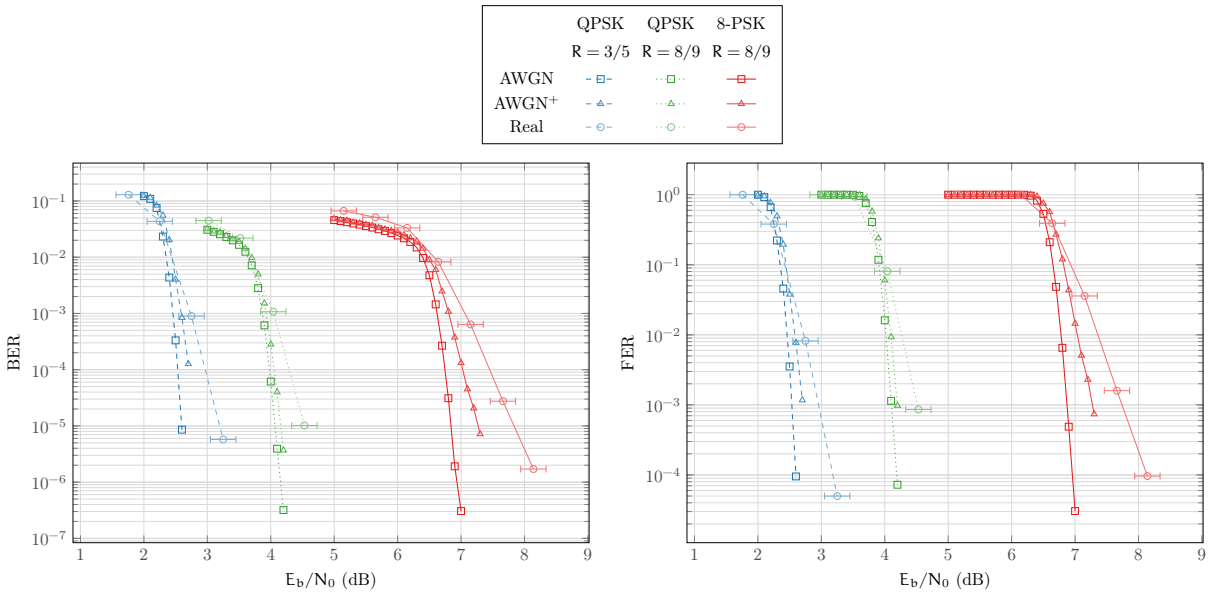


Figure 5.8 – DVB-S2 BER/FER decoding performance (LDPC BP h-layered, min-sum, 10 ite.).

5.4.3 Evaluation

This section evaluates the receiver part of the system. We did not bench the transmitter part as it is not the most compute intensive part and high throughputs are much more easier to reach. All the presented results have been obtained by running the code on a high-end machine composed by two Intel® Xeon™ Platinum 8168 CPUs and 128 GB of RAM. The frequency of the CPUs is 2.70 GHz and the *Turbo Boost* mode has been disabled for the reproducibility of the results. Each CPU is composed by 24 cores. They support the SMT technology but we disabled it in our tests to get the lowest possible tasks latency per core. 48 cores (or hardware threads) are available for the DVB-S2 receiver. All input and output data are represented by 32-bit floating-point or integer numbers. Knowing that the Platinum 8168 CPU supports the AVX-512 ISA. Sixteen 32-bit elements can be processed in one SIMD instruction. Most of the receiver tasks have been accelerated with MIPP SIMD functions. By this way, MIPP has even been enriched with new functions to improve the complex numbers support.

Knowing that the LDPC decoding is one of the most compute intensive task of the receiver, we decided to use the efficient inter-frame SIMD implementation presented before in the document (the early termination criterion has been switched on). This choice has the effect of computing sixteen frames at once in each task of the receiver. As discussed before, it negatively affects the overall latency of the system (by a factor of sixteen). But it is not important in the video streaming targeted application. The *Decoder LDPC* task (t_{16}^{Rx}) is the only one in the receiver to take advantage of the inter-frame SIMD technique. The other tasks simply process sixteen frames sequentially.

Table 5.2 presents the tasks throughputs and latencies measured for a sequential execution of the MODCOD 2 in the transmission phase. The tasks have been regrouped per stage in order to introduce the future decomposition when the parallelism is applied. The throughput is given in mega samples per second (MS/s). Indeed, some of the tasks are not working on bits but on samples and a bit is the smallest possible sample. The average (Avg), minimum (Min.) and maximum (Max.) throughputs are calculated with the number of output samples. Depending on the task, the number of output samples can drastically vary. Consequently, it is

not possible to directly compare the task throughputs with each others. This is the normalized average throughput ($\mathcal{N}_{\text{Avg}}$) has been introduced. It is the average throughput considering the K information bits independently of the task output socket size. In a first observation, one may notice that the latencies of the tasks are very high compared to the ones presented in the simulator evaluation (see Section 4.6). This is mainly due to the very large frame size N = 16740 and to the inter-frame level. Indeed, each task processes approximatively 16740 × 16 samples when in the simulator evaluation each task processes approximatively 2048 bits (there is a factor of ≈ 130). The applicative context is very different and it is directly observable from the resulting latencies.

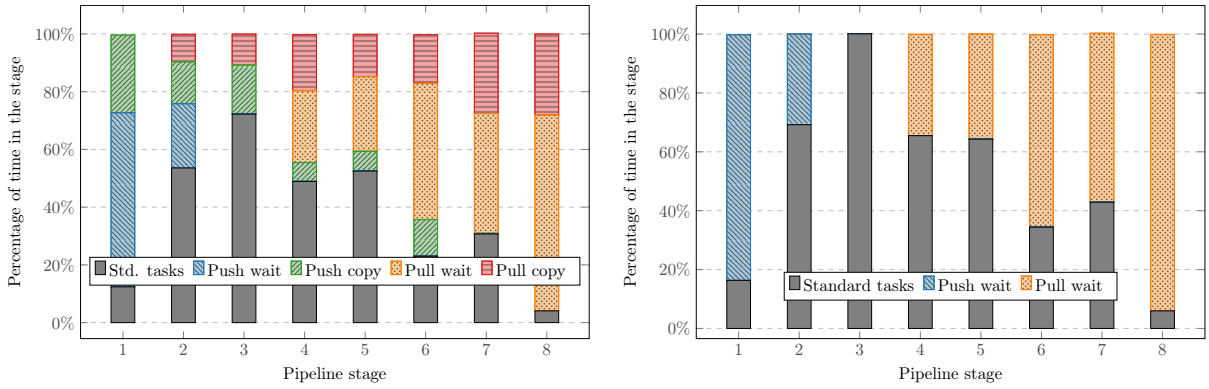
Table 5.2 – Tasks sequential throughputs and latencies of the DVB-S2 receiver (transmission phase, 16288 frames, inter-frame level = 16, MODCOD 2, error-free SNR zone). The sequential tasks are represented by orange rows. The slowest *sequential* stage is in red. The slowest of all stages is in blue.

Stages and Tasks	Throughput (MS/s)				Latency (µs)			Time
	Avg	Min.	Max.	$\mathcal{N}_{\text{Avg}}$	Avg	Min.	Max.	(%)
Radio - *receive* (t_1^{Rx})	1015.86	234.20	1093.98	431.83	527.32	489.66	2287.24	0.94
Stage 1	**1015.86**	**234.20**	**1093.98**	**431.83**	**527.32**	**489.66**	**2287.24**	**0.94**
Multiplier AGC - *imultiply* (t_2^{Rx})	864.41	420.05	935.71	367.45	619.71	572.49	1275.28	1.11
Synch. Freq. Coarse - *synchronize* (t_3^{Rx})	1979.17	665.98	2237.38	841.32	270.66	239.42	804.35	0.48
Filter Matched - *filter* (t_4^{Rx})	273.85	121.60	275.25	116.41	1956.08	1946.13	4405.09	3.49
Stage 2	**188.19**	**82.61**	**194.22**	**80.00**	**2846.45**	**2758.04**	**6484.72**	**5.08**
Synch. Timing - *synchronize* (t_5^{Rx})	130.38	58.97	131.31	55.42	4108.52	4079.39	9084.64	7.34
Stage 3	**130.38**	**58.97**	**131.31**	**55.42**	**4108.52**	**4079.39**	**9084.64**	**7.34**
Synch. Timing - *extract* (t_6^{Rx})	331.50	151.54	354.62	281.83	807.97	755.28	1767.48	1.44
Multiplier AGC - *imultiply* (t_7^{Rx})	806.31	442.69	877.19	685.51	332.18	305.34	605.02	0.59
Synch. Frame - *synchronize* (t_8^{Rx})	187.50	120.17	193.25	159.41	1428.51	1386.01	2228.76	2.55
Stage 4	**104.27**	**58.21**	**109.47**	**88.65**	**2568.66**	**2446.63**	**4601.26**	**4.58**
Scrambler Symbol - *descramble* (t_9^{Rx})	1979.41	668.85	2649.55	1682.89	135.31	101.09	400.45	0.24
Synch. Freq. Fine L&R - *synchronize* (t_{10}^{Rx})	1466.55	596.19	1741.72	1246.85	182.63	153.78	449.25	0.33
Synch. Freq. Fine P/F - *synchronize* (t_{11}^{Rx})	132.40	62.59	140.88	112.56	2022.98	1901.24	4279.30	3.61
Stage 5	**114.42**	**52.22**	**124.22**	**97.27**	**2340.92**	**2156.11**	**5129.00**	**4.18**
Framer PLH - *remove* (t_{12}^{Rx})	1148.07	427.71	1180.59	1008.60	225.77	219.55	606.02	0.40
Noise Estimator - *estimate* (t_{13}^{Rx})	626.12	151.24	656.09	550.06	413.98	395.07	1713.87	0.74
Stage 6	**405.16**	**111.73**	**421.72**	**355.94**	**639.75**	**614.62**	**2319.89**	**1.14**
Modem PSK - *demodulate* (t_{14}^{Rx})	46.07	42.12	46.28	40.47	5626.34	5600.83	6153.50	10.05
Interleaver - *deinterleave* (t_{15}^{Rx})	1533.54	518.95	1582.97	1347.25	169.02	163.74	499.47	0.30
Decoder LDPC - *decode SIHO* (t_{16}^{Rx})	166.15	69.12	171.59	164.21	1386.74	1342.74	3333.34	2.48
Decoder BCH - *decode HIHO* (t_{17}^{Rx})	6.92	6.15	6.96	6.92	32905.37	32705.15	36998.15	58.79
Scrambler Binary - *descramble* (t_{18}^{Rx})	91.11	47.74	91.73	91.11	2499.41	2482.41	4770.24	4.47
Stage 7	**5.35**	**4.40**	**5.38**	**5.35**	**42586.88**	**42294.87**	**51754.70**	**76.09**
Sink Binary File - *send* (t_{19}^{Rx})	1838.31	25.30	2100.47	1838.31	123.87	108.41	9001.34	0.22
Stage 8	**1838.31**	**25.30**	**2100.47**	**1838.31**	**123.87**	**108.41**	**9001.34**	**0.22**
Total	**4.09**	**2.51**	**4.14**	**4.09**	**55742.37**	**54947.73**	**90662.79**	**99.57**

The stage 7 takes 76% of the time with especially the *Decoder BCH* task (t_{17}^{Rx}) that takes 59% of the time. t_{17}^{Rx} should not take so many time compared to the other tasks. Indeed, we chose to not spend too much time in optimizing the BCH decoding process as the stage 7 throughput can easily be increased with the sequence duplication technique. The second slower stage in the stage 3. This stage is the main hotspot of the implemented receiver. The stage 3 contains only one synchronization task (t_5^{Rx}). In the current implementation this task cannot be duplicated (or parallelized) because there is an internal data dependency with the previous frame (state-full task). The stage 3 is the real limiting factor of the receiver. If a machine with an infinite number of cores is considered, the maximum reachable information throughput is 55.42 Mb/s.

We did not try to parallelize the waiting and the learning phases. We measured that the whole learning phase (1, 2 and 3) takes about one second. During the learning phase, the receiver is not fast enough to process the received samples in real time. To fix this problem, the samples are buffered in the *Radio - receive* task (t_1^{Rx}). Once the learning phase is done, the transmission phase is parallelized. Thus, the receiver becomes fast enough to absorb the radio buffer and samples in real time. During the transmission phase, the receiver is split into 8 stages as presented in Figure 5.7b. This decomposition has been motivated by the nature of the tasks (sequential or parallel) and by the sequential measured throughput. The number of stages has been minimized in order to limit the pipeline overhead. Consequently, sequential and parallel tasks have been regrouped in common stages. The slowest sequential task (t_5^{Rx}) has been isolated in the dedicated stage 3. The other sequential stages have been formed to always have a higher normalized throughput than the stage 3. The sequential throughput of the stage 7 (5.35 Mb/s) is lower than the throughput of the stage 3 (55.42 Mb/s). This is why the sequence duplication has been applied. The stage 7 has been parallelized over 28 threads. This looks overkill but the machine was dedicated to the DVB-S2 receiver and the throughput of the *Decoder LDPC* task (t_{16}^{Rx}) varies depending on the SNR. One can notice that an early termination criterion was enabled. When the signal quality is very good, the *Decoder LDPC* task runs fast and the threads can spend a lot of time in waiting. With the passive waiting version of the adaptor *push* and *pull* tasks, the CPU dynamically adapt the cores charge and energy can be saved. In Table 5.2, the presented *Decoder LDPC* task throughputs and latencies are optimistic because we are in a SNR error-free zone. All the threads are pinned to a single core with the *hwloc* library. The 28 threads of the stage 7 are pinned in round-robin between the CPU sockets. By this way, the memory bandwidth is maximized thanks to the two NUMA memory banks. The strategy of the stage 7 parallelism is to maximize the throughput. During the duplication process (modules clones), the thread pinning is known and the memory is copied into the right memory bank (first touch policy). All the other pipeline stages (1, 2, 3, 4, 5, 6 and 8) are running on a single thread. Because of the synchronizations between the pipeline stages (adaptor pushes and pulls), the threads have been pinned on the same socket. The idea is to minimize the pipeline stage latencies in maximizing the CPU cache performance. It avoids the extra-cost of moving the cache data between the sockets.

The receiver program needs around 1.3 GB of the global memory when running in sequential while it needs around 30 GB in parallel. The memory usage increases because of the sequence duplications in the stage 7. The duplication operation takes about 20 seconds. It is made at the very beginning of the program. It worth mentioning that the amount of memory was not a critical resource. So, we did not try to reduce its overall occupancy.



(a) Data copy (stage throughput is 40 Mb/s).

(b) Pointer copy (stage throughput is 55 Mb/s).

Figure 5.9 – Comparison of the two pipeline implementations in the receiver (MODCOD 2).

Figure 5.9 presents the repartition of the time in the pipeline stages (MODCOD 2). The receiver is running over 35 threads. Figure 5.9a shows the pipeline implementation with data copies. Figure 5.9b shows the pipeline implementation with pointer copies. *Push wait* and *Pull wait* are the percentage of time spent in passive or active waiting. *Push copy* and *Pull copy* are the percentage of time spent in copying the data to and from the adaptors buffers. *Standard tasks* is the cumulative percentage of time spent by the tasks presented in Figure 5.7b. In both implementations the pipeline stage throughput is constraint by the slowest one. In Figure 5.9a the measured throughput per stage is 40 Mb/s whereas in Figure 5.9b the measured throughput is 55 Mb/s. The pointer copy implementation throughput is $\approx 27\%$ higher than the data copy implementation. Figure 5.9a shows that the copy overhead is non-negligible. A 27% slowdown is directly due to these copies in the stage 3. It largely justifies the pointer copy implementation. In Figure 5.9b and in the stage 3, 100% of time is taken by the $\mathsf{t}_5^{\mathrm{Rx}}$ task. This is also confirmed by the measured throughput (55 Mb/s) which is very close the sequential throughput (55.42 Mb/s) reported in Table 5.2.

Table 5.3 – Throughput performance depending of the selected DVB-S2 configuration.

Config.	**Throughput** (Mb/s)			
	Sequential		**Parallel**	
	Info.	**Coded**	**Info.**	**Coded**
MODCOD 1	3.4	5.7	37	62
MODCOD 2	4.1	4.6	55	62
MODCOD 3	4.0	4.5	80	90

Table 5.3 summarizes the obtained throughputs for the 3 MODCODs presented in Table 5.1. Each time, sequential and parallel throughput are given. To measure to maximum achievable throughput, the USRP modules has been removed and replaced by the read of samples pre-registered in a binary file. This is because the pipeline stages are naturally adapting to the slowest one. It means that in a real communication, the throughput of the radio is always configured to be just a little bit slower than the slowest stage. Indeed, it is necessary for real time communication otherwise the radio task has to indefinitely bufferize the samples while the amount of available memory in the machine is clearly not infinite. The information throughput (K bits) is the final useful throughput for the user while the coded (N) throughput is here for observations. Between the MODCOD 1 and 2, only the LDPC code rate varies ($\mathsf{R} = 3/5$ and $\mathsf{R} = 8/9$ resp.). In the parallel implementation, it has a direct impact on the information throughput while the coded throughput is unchanged. Indeed, the *Decoder LDPC* task ($\mathsf{t}_{16}^{\mathrm{Rx}}$) is parallelized in the stage 7. This stage is capable to adapt to the charge automatically. In the sequential implementation, the coded throughput is negatively impacted when $\mathsf{R} = 8/9$. Between the MODCOD 2 and 3, the modulation varies (QPSK and 8-PSK resp.) and the frames have to be deinterleaved (column/row interleaver). High order modulation reduces the amount of samples processed by in the *Synchronizer Timing* task ($\mathsf{t}_5^{\mathrm{Rx}}$): this results in higher throughput (80 Mb/s for the 8-PSK) in the slowest stage 3. In the parallel implementation, the pipeline stage throughputs are adapting to the slowest stage 3. It results in an important speedup. In the sequential implementation, it results in a little slowdown. Indeed, the additional time spent in the *deinterleave* task ($\mathsf{t}_{15}^{\mathrm{Rx}}$) is higher than the time saved in the *Synchronizer Timing* task ($\mathsf{t}_5^{\mathrm{Rx}}$).

These results demonstrate the benefit of the parallelized implementation of the proposed receiver. The throughput speedup ranges from 10 to 20 compared to the sequential implementation. It is also important to note that the selected configurations are efficient in different SNR zones (as shown in Figure 5.8). Depending on the signal quality, different MODCOD can be selected.

A direct impact in observed on the system throughput. For instance, MODCOD 1 is adapted for noisy environments (3 dB). However the information throughput is limited to 37 Mb/s while MODCOD 3 is more adapted to clearer signal conditions (7.5 dB) and the information throughput reaches 80 Mb/s. MODCOD 2 is in-between.

5.4.4 Related Works

Some other works are focusing on the SDR implementation of a DVB-S2 transceiver. To the best of our knowledge, here is a list of the existing projects:

— **gr-dvbs2rx** [Eco18] is an open source extension to GNU Radio. The project sounds promising but lacks efficiency. Its main maintainer affirms that the receiver is not yet able to meet the satellite real time constraints (30 to 50 Mb/s) on a Xeon™ Gold/Platinum series processor [2]. It advises to use a dedicated GPU or FPGA for the LDPC decoding.

— **leansdr** [pab16] is a standalone open source project. The project creation was motivated to reach higher receiver throughput than GNU Radio even if it results in decoding performance degradations. For instance, a low complexity LDPC bit-flipping decoder [RL09] is chosen. At the time of the writing, the project does not support multi-threading and SIMD instructions.

— **Grayver and Utter** recently published a paper [GU20] in which they succeed to build a 10 Gb/s DVB-S2 receiver on a cluster of server-class CPUs. On a comparable CPU, their work is able to double or even triple the throughput of our implementation. This is mainly due to the use of a high speed SIMD LDPC decoder [LJ16, Gra19] and to new algorithmic improvements in the synchronization tasks. For instance, they were able to express more parallelism than us in the *Synchronizer Timing* task (t_5^{Rx}). It is very promising. However, we also tried some aggressive optimizations in the *Synchronizer Timing* task but we never succeeded to keep the same level of BER/FER decoding performance. It could be interesting to check if the Grayver and Utter work comes with no penalty in terms of decoding performance and to combine their optimizations with the AFF3CT eDSL. Unlike our work, Grayver and Utter are focusing on a single DVB-S2 MODCOD (8-PSK, $\mathsf{N} = 64800$ and $\mathsf{R} = 1/2$).

5.5 Conclusion

In this chapter, a new eDSL dedicated to the SDR has been presented. Main components have been designed to satisfy the SDR needs in terms of 1) expressiveness with sequences, tasks and loops; 2) performance with the sequence duplication technique and the pipelining strategy. We evaluated the proposed eDSL in an applicative context: the software implementation of the DVB-S2 standard. The results demonstrate the efficiency of the AFF3CT eDSL. Indeed, the proposed solution matches satellite real time constraints ($30 \sim 50$ Mb/s). This is the consequence of two main factors: 1) the task level optimizations, for instance a fast LDPC decoder has been applied (see Chapter 2); 2) the quasi zero overhead eDSL, with among others, an efficient implementation of the pipeline technique.

In future works, it could be interesting to combine the parallel features of the AFF3CT eDSL with high level languages like Python or MATLAB®. The signal processing community is often not familiar with the `C++` language. They could use the existing high-speed `C++` tasks and develop their own in the high level language. Linking with the AFF3CT library, it would

2. https://lists.gnu.org/archive/html/discuss-gnuradio/2019-01/msg00196.html.

be convenient if the tasks written in the high level language could be inserted in the AFF3CT sequences. This way the whole system would automatically be parallelized. This is technically possible in our model. As the static scheduling is evaluated at the runtime, it is possible to encapsulate a function from a high level language in AFF3CT task (with a callback) and to start the sequence analysis (= perform the static scheduling resolution) after that. In comparison, it would be more complicated to make the languages interfacing with GNU Radio as the static scheduling is resolved at the compilation time and flatten in the Python language. When the generated Python code is executed, it performs calls to the tasks written and compiled in a `C++` library. Moreover, it could be very useful to propose a graphical user interface like *GNU Radio Companion* to facilitate the tasks creation and binding as well as the parallelism definition (sequence duplication, pipeline stages). The graphical user interface would then generate a ready to compile `C++` code similar to the one presented in Listing 5.1.

Conclusions and Perspectives

Conclusion

In the context of digital communications, channel coding schemes are widely spread. This thesis focuses on three channel codes that are present in most of the current digital communication standards: the LDPC codes, the polar codes and the turbo codes. In digital communication systems, most of the computational time is spent in the receiver and more precisely in the decoding stage. This is why, we propose efficient implementations of these decoding algorithms on CPUs. The proposed implementations enable fast evaluations and validations of various configurations. Moreover, there is a growing need to build full digital communication chains in software. This is what we call the Software-Defined Radio (SDR). Thus, the challenge is to take advantage of multi-core CPU architectures to schedule the processing in parallel.

Several optimization strategies have been presented and discussed. One of the main characteristic of the digital communication algorithms is that they have a very short execution time (low latency). Thus, the most adapted parallelism level presents in the actual CPUs is the Single Instruction Multiple Data (SIMD) model. In Section 2.1, MIPP, a generic SIMD library, is proposed. This library enables simplified and portable use of the CPUs vectorized instructions. Then, in Section 2.2, two main vectorization strategies are detailed: the intra-frame SIMD strategy that enables very low latency implementations and the inter-frame SIMD strategy that enables very high throughput implementations. The intra-frame SIMD strategy consists in using the algorithm inherent parallelism to speedup the computation in a single frame while the inter-frame SIMD strategy processes several frames in parallel. In a second part of this chapter, specific optimizations for each channel codes are given. First, a new SIMD implementation of the LDPC Belief Propagation (BP) decoder is proposed. This decoder rests upon the inter-frame strategy and focuses on maximizing the flexibility. Indeed, it is able to adapt to many algorithmic sub-variants which is without precedent in the domain. Then, the optimizations of two polar decoders are proposed, namely the Successive Cancellation (SC) and the Successive Cancellation List (SCL) algorithms. Both the inter-frame and intra-frame strategies are implemented. This two decoders are based on a recursive description and the decoding process can be seen as a tree traversal. Some specific optimizations like the tree pruning are performed to drastically reduce the number of tree nodes. The recursive calls have also been unrolled and generated decoders are proposed to reach the best possible throughputs and latencies. This comes at the cost of reduced flexibility. Finally, an SIMD implementation of the turbo decoder (max-log-MAP algorithms) is given. The implementation uses the inter-frame SIMD strategy and targets high throughputs. Specific optimizations have been made to increase the decoder efficiency: some loops at the core of the decoding process have been merged and unrolled to increase the registers reuse.

AFF3CT is a library of digital communication algorithms, developed as part of this thesis, focusing on high performance implementations. Its software architecture supports the algorithmic heterogeneity. Many channel codes are supported like the LDPC codes, the polar codes and the

turbo codes detailed before. To the best of our knowledge, AFF3CT is the library with the most comprehensive support for channel coding algorithms. The toolbox also includes a BER/FER simulator. Many digital communication systems can be evaluated over various parameters. The simulator takes advantage of the multi-core CPU architectures to reduce the restitution time. All these features have been designed to enable reproducible science. A BER/FER comparator tool has been developed to easily search in a database of 500 pre-simulated BER/FER references. All there references are results simulated with AFF3CT and can be reproduced. To this purpose, a pipeline of tests has been implemented. Each time there is a modification in the source code, the database of references is replayed to avoid regressions.

The new implementations have been evaluated and compared with the state-of-the-art. The results show levels of performance close to the best software implementations in the literature. Exhaustive surveys are given through software decoder Hall of Fames (HoFs). The proposed decoders are reported as well as state-of-the-art works. These HoFs enable to compare CPU and GPU implementations. Some metrics like the normalized throughput, the Throughput Under Normalized Decoding Cost (TNDC) and the energy consumption are defined. Finally the AFF3CT simulator performance is evaluated over several server-class CPUs. It shows that the simulator is able to take advantage of various SIMD instructions and multi-core architectures. During the simulation of a polar code, a peak performance of 11 Gb/s is reached on a AMD® EPYC CPU. To the best of our knowledge, this is the first work to reach this level of performance.

The AFF3CT library has been enriched with a new embedded Domain Specific Language (eDSL). The main components have been designed to satisfy the SDR needs in terms of 1) expressiveness; 2) performance. Most of the digital communication systems can be represented by a directed graph of processing tasks (dataflow model). The proposed eDSL uses this representation to improve the expressiveness. Indeed, the tasks data transfers and their execution are automatically managed by the eDSL. Moreover, the performance is an critical aspect. To reduce the execution time, some data independent parts of the graph of tasks can be duplicated. Each duplication can be executed on separated CPU cores. This strategy leads to an increased throughput. However, when it cannot be applied the well-known pipeline strategy have been implemented. Thus, the performance of the overall communication system can be increased up to the throughput of the slowest task. Then, the proposed eDSL is evaluated in an applicative context: the software implementation of the DVB-S2 standard physical layer. The results demonstrate the efficiency of the AFF3CT eDSL. Indeed, the proposed solution matches satellite real-time constraints (30 $\sim$ 50 Mb/s). This is the consequence of two main factors: 1) the task level optimizations, 2) the low overhead eDSL, with among others, an efficient implementation of the pipeline technique.

AFF3CT is currently used in several industrial contexts for simulation purposes (Turbo concept, Airbus, Thales, Huawei) and for specific developments (CNES, Schlumberger, Airbus, Thales, Orange, Safran), as well as in academic projects (NAND French National Agency project, IdEx CPU, R&T CNES). The MIT license chosen for the project enables industrial and academic partners to reuse parts of AFF3CT in their own projects without any restriction. Moreover, AFF3CT has been cited in scientific publications. Many works are exploiting the AFF3CT simulator as a reference for the decoding performance. In other works, AFF3CT has been enriched to support new features. And, in some cases, AFF3CT is used as a library where some sub-parts of the toolbox are reused or some methodologies are extracted.

To conclude on this thesis work, the main contributions are 1) the definition of task level optimization techniques that enable high performance portable implementations of signal processing algorithms on CPUs, 2) an open-source software that enables homogeneous uses of various

algorithms and implementations and 3) a new language dedicated to the SDR needs that enables to define digital communication systems taking advantage of the CPUs parallel architecture. AFF3CT has been designed for high performance keeping in mind that the algorithms come from the signal community experts that are not familiar with CPU optimization techniques. Consequently, there is a clear separation of concerns between the tasks design and their parallel execution. Co-design is then possible: signal experts can focus on the tasks description while HPC experts can work independently on the parallel execution thanks to the eDSL abstraction. To the best of our knowledge, AFF3CT is the first environment to propose this level of performance combined with the integration of many digital communication algorithms.

Perspectives

Several study and research perspectives remain to be explored following this thesis work. A non-exhaustive list of these perspectives is given below. This list is given in ascending order of presupposed complexity.

First, thanks to the flexibility of the proposed software architecture, new coding schemes can be explored. The channel coding theory is constantly evolving and it is mandatory to be able to evaluate the performance of new schemes. For instance, the polar codes are one of the main interest in the domain. They have been recently generalized from their discovery by Arıkan. It is possible to build new codes from various kernels that are not just powers of two. This is called multi-kernel polar codes. Some preliminary works have been conducted to find kernels that have good factorization properties. However, this is a brute force exploration and the complexity grows exponentially with the size of the kernels. It could be interesting to reduce the kernel exploration domain and to apply HPC techniques to reduce the finding time. The multi-kernel polar codes construction is a promising area of research that could lead to better finite-length decoding performance.

One of the main contribution of this thesis is to propose efficient digital signal processing methods and implementations on CPU. Nowadays there is a growing interest for GPUs in the HPC community. The GPUs are very parallel architectures. In some conditions, the GPU implementations can lead to non-negligible reduction of the computational time compared to the implementations on CPU. It could be interesting to study the integration of GPU tasks in AFF3CT. One of the main challenges is to manage the CPU to GPU and GPU to CPU transfers. On GPUs, many works are focusing on implementing only the most compute intensive task (namely the channel decoder) or a fixed configuration of tasks (BPSK modulation, AWGN channel and a specific coding scheme) [Wu+11, Xia+13, LJC14, Lai+16, Gia+16, KK17b]. The several configurations available in AFF3CT combined with the ability to execute tasks on both CPU and GPU would be a major improvement. Even if the GPUs are a good alternative to the CPUs, we believe that they will not be integrated in C-RAN architectures. The FPGAs look like to provide a better compromize between power efficiency and computational performance for scaling up. Their integration in AFF3CT could be a great challenge.

Finally, the proposed eDSL could be enriched. For instance, the pipeline stages are given by the user while they could be found automatically. The execution time of the tasks is mostly constant for a given CPU. Thus, an auto-tuning phase could be applied to determine a good configuration of the pipeline stages automatically. Moreover, the scheduling of the tasks inside a sequence is very basic. The tasks are not executed in parallel even if the data dependencies allow it. We think that a dynamic scheduling strategy like it can be found in the HPC runtime

libraries (see OpenMP or StarPU) would be overkill. The overhead of a dynamic scheduler is not negligible because the execution time of the signal processing tasks in very short (ranging from some nanoseconds to some microseconds). However, an improved static scheduling strategy that enables parallel executions inside sequences would certainly help to reduce the restitution time. These improvements could lead to an extension of the AFF3CT eDSL. Indeed, the targeted domain could be expended to the generalized streaming applications (image/video processing, cryptographic processing, networking, DSP, etc.). The challenge will be to identify the required additional modules and to integrate them into the eDSL with no impact on the execution efficiency.

Bibliography

[Ack82] W.B. Ackerman. "Data Flow Languages". In: *IEEE Computer* 15 (Feb. 1982), pp. 15–25. DOI: 10.1109/MC.1982.1653938 (cit. on p. 115).

[AD18] R. Akeela and B. Dezfouli. "Software-Defined Radios: Architecture, State-of-the-Art, and Challenges". In: *ACM Computer Communications* 128 (2018), pp. 106–125. DOI: 10.1016/j.comcom.2018.07.012 (cit. on p. 115).

[aic18] aicodix GmbH. *Reusable C++ Not-DSP-related Code Library*. 2018. URL: https://github.com/aicodix/code (cit. on p. 72).

[AK11] A. Alamdar-Yazdi and F. R. Kschischang. "A Simplified Successive-Cancellation Decoder for Polar Codes". In: *IEEE Communications Letters (COMML)* 15.12 (Dec. 2011), pp. 1378–1380. DOI: 10.1109/LCOMM.2011.101811.111480 (cit. on p. 16).

[Alt15a] Altera Innovate Asia FPGA Design Contest. *5G Algorithm Innovation Competition*. 2015. URL: http://www.innovateasia.com/5g/en/gp2.html (visited on 09/16/2018) (cit. on p. 26).

[Alt15b] Altera University Program. *The 1st 5G Algorithm Innovation Competition-SCMA*. 2015. URL: http://www.innovateasia.com/5g/images/pdf/1st%205G%20Algorithm%20Innovation%20Competition-ENV1.0%20-%20SCMA.pdf (cit. on p. 78).

[Aly+19] R. M. Aly, A. Zaki, W. K. Badawi, and M. H. Aly. "Time Coding OTDM MIMO System Based on Singular Value Decomposition for 5G Applications". In: *MDPI Applied Sciences* 9.13 (July 2019), p. 2691. DOI: 10.3390/app9132691 (cit. on p. 85).

[Ama+05] S. Amarasinghe, M. l. Gordon, M. Karczmarek, J. Lin, D. Maze, R. M. Rabbah, and W. Thies. "Language and Compiler Design for Streaming Applications". In: *Springer International Journal of Parallel Programming (IJPP)* 2.33 (June 2005), pp. 261–278. DOI: 10.1007/s10766-005-3590-6 (cit. on p. 115).

[Arı09] E. Arıkan. "Channel Polarization: A Method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels". In: *IEEE Transactions on Information Theory (TIT)* 55.7 (July 2009), pp. 3051–3073. DOI: 10.1109/TIT.2009.2021379 (cit. on pp. 9, 14).

[Arı11] E. Arıkan. "Systematic Polar Coding". In: *IEEE Communications Letters (COMML)* 15.8 (Aug. 2011), pp. 860–862. DOI: 10.1109/LCOMM.2011.061611.110862 (cit. on p. 13).

[Arm+16] B. Armstrong, L. Teske, P. Noordhuis, T. Petazzoni, J. Carlson, and E. Betts. *libcorrect*. 2016. URL: https://github.com/quiet/libcorrect (cit. on p. 72).

[ARS81] T. Aulin, N. Rydbeck, and C. Sundberg. "Continuous Phase Modulation - Part II: Partial Response Signaling". In: *IEEE Transactions on Communications (TCOM)* 29.3 (Mar. 1981), pp. 210–225. DOI: 10.1109/TCOM.1981.1094985 (cit. on p. 78).

[AS81] T. Aulin and C. Sundberg. "Continuous Phase Modulation - Part I: Full Response Signaling". In: *IEEE Transactions on Communications (TCOM)* 29.3 (Mar. 1981), pp. 196–209. DOI: 10.1109/TCOM.1981.1095001 (cit. on p. 78).

[Bah+74] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv. "Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate (Corresp.)" In: *IEEE Transactions on Information Theory (TIT)* 20.2 (Mar. 1974), pp. 284–287. DOI: 10.1109/TIT.1974.1055186 (cit. on p. 20).

[Ban+14] S. Bang, C. Ahn, Y. Jin, S. Choi, J. Glossner, and S. Ahn. "Implementation of LTE System on an SDR Platform using CUDA and UHD". In: *Springer Journal of Analog Integrated Circuits and Signal Processing (AICSP)* 78.3 (Mar. 1, 2014), p. 599. DOI: 10.1007/s10470-013-0229-1 (cit. on p. 115).

[BD10] D. Black-Schaffer and W. J. Dally. "Block-Parallel Programming for Real-Time Embedded Applications". In: *International Conference on Parallel Processing (ICPP)*. IEEE, Sept. 2010, pp. 297–306. DOI: 10.1109/ICPP.2010.37 (cit. on p. 115).

[BGT93] C. Berrou, A. Glavieux, and P. Thitimajshima. "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes". In: *International Conference on Communications (ICC)*. Vol. 2. IEEE, May 1993, pp. 1064–1070. DOI: 10.1109/ICC.1993.397441 (cit. on p. 8).

[Bil+95] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. "Cyclo-Static Data Flow". In: *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Vol. 5. IEEE, May 1995, pp. 3255–3258. DOI: 10.1109/ICASSP.1995.479579 (cit. on p. 115).

[BM58] G. E. P. Box and M. E. Muller. "A Note on the Generation of Random Normal Deviates". In: *The Annals of Mathematical Statistics* 29.2 (1958), pp. 610–611. DOI: 10.1214/aoms/1177706645 (cit. on p. 42).

[BPB15] A. Balatsoukas-Stimming, M. B. Parizi, and A. Burg. "LLR-Based Successive Cancellation List Decoding of Polar Codes". In: *IEEE Transactions on Signal Processing (TSP)* 63.19 (Oct. 2015), pp. 5165–5179. DOI: 10.1109/TSP.2015.2439211 (cit. on p. 15).

[BR60] R.C. Bose and D.K. Ray-Chaudhuri. "On a Class of Error Correcting Binary Group Codes". In: *Elsevier Information and Control* 3.1 (1960), pp. 68–79. DOI: 10.1016/S0019-9958(60)90287-4 (cit. on p. 8).

[Bro+10] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. "hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications". In: *Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE, Feb. 2010, pp. 180–186. DOI: 10.1109/PDP.2010.67 (cit. on p. 123).

[Buc+04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. "Brook for GPUs: Stream Computing on Graphics Hardware". In: *ACM Transactions on Graphics (TOG)* 23.3 (Aug. 2004), pp. 777–786. DOI: 10.1145/1015706.1015800 (cit. on p. 115).

[Cam+17] S. Cammerer, B. Leible, M. Stahl, J. Hoydis, and S. ten Brink. "Combining Belief Propagation and Successive Cancellation List Decoding of Polar Codes on a GPU Platform". In: *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, Mar. 2017, pp. 3664–3668. DOI: 10.1109/ICASSP.2017.7952840 (cit. on p. 104).

[CC04] P. Coulton and D. Carline. "An SDR Inspired Design for the FPGA Implementation of 802.11a Baseband System". In: *International Symposium on Consumer Electronics (ISCE)*. IEEE, Sept. 2004, pp. 470–475. DOI: 10.1109/ISCE.2004.1375991 (cit. on p. 115).

[Cen19] T. Cenova. "Exploring HLS Coding Techniques to Achieve Desired Turbo Decoder Architectures". PhD thesis. Rochester Institute of Technology, 2019. URL: https://scholarworks.rit.edu/theses/10256/ (cit. on p. 85).

[CF02] J. Chen and M. P. C. Fossorier. "Density Evolution for Two Improved BP-Based Decoding Algorithms of LDPC Codes". In: *IEEE Communications Letters (COMML)* 6.5 (May 2002), pp. 208–210. DOI: 10.1109/4234.1001666 (cit. on p. 12).

[Cha+11] C. C. Chang, Y. L. Chang, M. Y. Huang, and B. Huang. "Accelerating Regular LDPC Code Decoders on GPUs". In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing (J-STARS)* 4.3 (Sept. 2011), pp. 653–659. DOI: 10.1109/JSTARS.2011.2142295 (cit. on p. 103).

[Cha72] D. Chase. "Class of Algorithms for Decoding Block Codes with Channel Measurement Information". In: *IEEE Transactions on Information Theory (TIT)* 18.1 (Jan. 1972), pp. 170–182. DOI: 10.1109/TIT.1972.1054746 (cit. on p. 57).

[Che+13] Xiang Chen, Ji Zhu, Ziyu Wen, Yu Wang, and Huazhong Yang. "BER Guaranteed Optimization and Implementation of Parallel Turbo Decoding on GPU". In: *International Conference on Communications and Networking in China (CHINACOM)*. IEEE, Aug. 2013, pp. 183–188. DOI: 10.1109/ChinaCom.2013.6694588 (cit. on pp. 59, 105).

[Che+15] A. Checko, H. L. Christiansen, Y. Yan, L. Scolari, G. Kardaras, M. S. Berger, and L. Dittmann. "Cloud RAN for Mobile Networks – A Technology Overview". In: *IEEE Communications Surveys Tutorials* 17.1 (2015), pp. 405–426. DOI: 10.1109/COMST.2014.2355255 (cit. on p. 25).

[Chr12] G. Chrysos. "Intel Xeon Phi coprocessor (codename Knights Corner)". In: *Hot Chips Symposium (HCS)*. IEEE, Aug. 2012, pp. 1–31. DOI: 10.1109/HOTCHIPS.2012.7476487 (cit. on p. 108).

[COP+05] B. Cristea, T. Ottosson, A. Piątyszek, et al. *IT++*. 2005. URL: http://itpp.sourceforge.net (cit. on p. 72).

[CS12] S. Chinnici and P. Spallaccini. "Fast Simulation of Turbo Codes on GPUs". In: *International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*. IEEE, Aug. 2012, pp. 61–65. DOI: 10.1109/ISTC.2012.6325199 (cit. on pp. 59, 105).

[CTG19a] A. Cavatassi, T. Tonnellier, and W. J. Gross. "Asymmetric Construction of Low-Latency and Length-Flexible Polar Codes". In: *International Conference on Communications (ICC)*. IEEE, 2019, pp. 1–6. DOI: 10.1109/ICC.2019.8761129 (cit. on p. 85).

[CTG19b] A. Cavatassi, T. Tonnellier, and W. J. Gross. "Fast Decoding of Multi-Kernel Polar Codes". In: *Wireless Communications and Networking Conference (WCNC)*. IEEE, 2019. DOI: 10.1109/WCNC.2019.8885698 (cit. on p. 85).

[Cun+09] M. Cunche, J. Detchart, J. Lacan, V. Roca, et al. *OpenFEC*. 2009. URL: http://openfec.org (cit. on p. 72).

[CWC15] M. Cheng, Y. Wu, and Y. Chen. "Capacity Analysis for Non-orthogonal Overloading Transmissions under Constellation Constraints". In: *International Conference on Wireless Communications Signal Processing (WCSP)*. IEEE, Oct. 2015, pp. 1–5. DOI: 10.1109/WCSP.2015.7341294 (cit. on p. 78).

[Deb+16a] I. Debbabi, B. Le Gal, N. Khouja, F. Tlili, and C. Jégo. "Real Time LP Decoding of LDPC Codes for High Correction Performance Applications". In: *IEEE Wireless Communications Letters (WCL)* 5.6 (Dec. 2016), pp. 676–679. DOI: 10.1109/LWC.2016.2615304 (cit. on pp. 85, 103).

[Deb+16b] I. Debbabi, N. Khouja, F. Tlili, B. Le Gal, and C. Jégo. "Multicore Implementation of LDPC Decoders based on ADMM Algorithm". In: *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, Mar. 2016, pp. 971–975. DOI: 10.1109/ICASSP.2016.7471820 (cit. on pp. 85, 103).

[Del+20] Y. Delomier, B. Le Gal, J. Crenne, and C. Jego. "Model-Based Design of Hardware SC Polar Decoders for FPGAs". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 13.2 (May 2020). DOI: 10.1145/3391431 (cit. on p. 85).

[Den80] J.B. Dennis. "Data Flow Supercomputers". In: *IEEE Computer* 13.11 (Nov. 1980), pp. 48–56. DOI: 10.1109/MC.1980.1653418 (cit. on p. 115).

[DLB17] P. De Oliveira Castro, S. Louise, and D. Barthou. "DSL Stream Programming on Multicore Architectures". In: *Programming Multi-core and Many-core Computing Systems*. John Wiley and Sons, 2017. Chap. 7. DOI: 10.1002/9781119332015.ch7 (cit. on p. 115).

[Duf20] Ken R. Duffy. *Ordered Reliability Bits Guessing Random Additive Noise Decoding*. Jan. 2020. arXiv: 2001.00546 [cs.IT] (cit. on p. 84).

[Dut+10] P. Dutta, Y. Kuo, A. Ledeczi, T. Schmid, and P. Volgyesi. "Putting the Software Radio on a Low-calorie Diet". In: *Workshop on Hot Topics in Networks (HotNets)*. ACM, 2010. DOI: 10.1145/1868447.1868467 (cit. on p. 115).

[Eco18] R. Economos. *gr-dvbs2rx: GNU Radio Extensions for the DVB-S2 and DVB-T2 Standards*. 2018. URL: https://github.com/drmpeg/gr-dvbs2rx (cit. on p. 131).

[Eli54] P. Elias. "Error-free Coding". In: *Transactions of the IRE Professional Group on Information Theory* 4.4 (Sept. 1954), pp. 29–37. DOI: 10.1109/TIT.1954.1057464 (cit. on p. 8).

[Eli55] P. Elias. "Coding for Two Noisy Chanels". In: *IRE Convention Record*. IEEE, Apr. 1955. URL: http://web.mit.edu/6.441/www/reading/hd2.pdf (cit. on pp. 8, 17).

[Eng+94] M. Engels, G. Bilsen, R. Lauwereins, and J. A. Peperstraete. "Cycle-Static Dataflow: Model and Implementation". In: *Asilomar Conference on Signals, Systems, and Computers (ACSSC)*. Vol. 1. IEEE, Oct. 1994, pp. 503–507. DOI: 10.1109/ACSSC.1994.471504 (cit. on p. 115).

[Erc+17] F. Ercan, C. Condo, S. A. Hashemi, and W. J. Gross. "On Error-correction Performance and Implementation of Polar Code List Decoders for 5G". In: *Allerton Conference on Communication, Control, and Computing*. Oct. 2017, pp. 443–449. DOI: 10.1109/ALLERTON.2017.8262771 (cit. on p. 85).

[Eri15] Ericsson. *Cloud RAN - The Benefits of Virtualization, Centralisation and Coordination*. Tech. rep. Ericsson, 2015. URL: https://www.ericsson.com/assets/local/publications/white-papers/wp-cloud-ran.pdf (cit. on p. 25).

[Est+12a] P. Estérie, M. Gaunard, J. Falcou, and J. T. Lapresté. "Exploiting Multimedia Extensions in C++: A Portable Approach". In: *IEEE Computing in Science & Engineering (CS&E)* 14.5 (Sept. 2012), pp. 72–77. DOI: 10.1109/MCSE.2012.96 (cit. on pp. 36, 37).

[Est+12b] P. Estérie, M. Gaunard, J. Falcou, J. T. Lapresté, and B. Rozoy. "Boost.SIMD: Generic programming for portable SIMDization". In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM/IEEE, Sept. 2012, pp. 431–432. DOI: 10.1145/2370816.2370881 (cit. on pp. 36, 37).

[ETG20] F. Ercan, T. Tonnellier, and W. J. Gross. "Energy-Efficient Hardware Architectures for Fast Polar Decoders". In: *IEEE Transactions on Circuits and Systems I: Regular Papers (TCAS1)* 67.1 (Jan. 2020), pp. 322–335. DOI: 10.1109/TCSI.2019.2942833 (cit. on p. 85).

[ETS05] ETSI. *EN 302 307 - Digital Video Broadcasting (DVB); Second Generation Framing Structure, Channel Coding and Modulation Systems for Broadcasting, Interactive Services, News Gathering and Other Broadband Satellite Applications (DVB-S2)*. Mar. 2005. URL: https://www.etsi.org/deliver/etsi_en/302300_302399/302307/01.02.01_60/en_302307v010201p.pdf (cit. on p. 124).

[ETS13] ETSI. *3GPP - TS 36.212 - Multiplexing and Channel Coding (R. 11)*. Aug. 2013. URL: https://www.etsi.org/deliver/etsi_ts/136200_136299/136212/11.03.00_60/ts_136212v110300p.pdf (cit. on p. 20).

[ETS18] ETSI. *3GPP - TS 38.212 - Multiplexing and Channel Coding (R. 15)*. Aug. 2018. URL: https://www.etsi.org/deliver/etsi_ts/138200_138299/138212/15.02.00_60/ts_138212v150200p.pdf (cit. on p. 115).

[EUR13] EURECOM. *OpenAirInterface (OAI)*. 2013. URL: https://gitlab.eurecom.fr/oai/openairinterface5g (cit. on p. 72).

[Fal+08] G. Falcao, V. Silva, L. Sousa, and J. Marinho. "High Coded Data Rate and Multi-codeword WiMAX LDPC Decoding on Cell/BE". In: *IET Electronics Letters* 44.24 (Nov. 2008), pp. 1415–1416. DOI: 10.1049/el:20081927 (cit. on p. 103).

[Fal+09] G. Falcão, S. Yamagiwa, V. Silva, and L. Sousa. "Parallel LDPC Decoding on GPUs Using a Stream-Based Computing Approach". In: *Springer Journal of Computer Science and Technology (JCST)* 24.5 (Sept. 1, 2009), p. 913. DOI: 10.1007/s11390-009-9266-8 (cit. on p. 103).

[Fal+11] G. Falcao, J. Andrade, V. Silva, and L. Sousa. "GPU-Based DVB-S2 LDPC Decoder with High Throughput and Fast Error Floor Detection". In: *IET Electronics Letters* 47.9 (Apr. 2011), pp. 542–543. DOI: 10.1049/el.2011.0201 (cit. on p. 103).

[Fal+12] G. Falcao, V. Silva, L. Sousa, and J. Andrade. "Portable LDPC Decoding on Multicores Using OpenCL". In: *IEEE Signal Processing Magazine* 29.4 (July 2012), pp. 81–109. DOI: 10.1109/MSP.2012.2192212 (cit. on p. 103).

[FAN07] T. Furtak, J. N. Amaral, and R. Niewiadomski. "Using SIMD Registers and Instructions to Enable Instruction-Level Parallelism in Sorting Algorithms". In: *Symposium on Parallel Algorithms and Architectures*. San Diego, California, USA: ACM, 2007, pp. 348–357. DOI: 10.1145/1248377.1248436 (cit. on p. 57).

[Flo18] F. Florian. "PHYSIM - A Physical Layer Simulation Software". In: *International Conference on Consumer Electronics (ICCE)*. IEEE, Sept. 2018, pp. 1–6. DOI: 10.1109/ICCE-Berlin.2018.8576187 (cit. on p. 85).

[FMI99] M. P. C. Fossorier, M. Mihaljevic, and H. Imai. "Reduced Complexity Iterative Decoding of Low-Density Parity Check Codes based on Belief Propagation". In: *IEEE Transactions on Communications (TCOM)* 47.5 (May 1999), pp. 673–680. DOI: 10.1109/26.768759 (cit. on p. 12).

[Fog17] A. Fog. *C++ Vector Class Library (VCL)*. 2017. URL: http://www.agner.org/optimize/#vectorclass (cit. on p. 36).

[For73] G. D. Forney. "The Viterbi Algorithm". In: *Proceedings of the IEEE* 61.3 (Mar. 1973), pp. 268–278. DOI: 10.1109/PROC.1973.9030 (cit. on p. 18).

[FSS11] G. Falcao, L. Sousa, and V. Silva. "Massively LDPC Decoding on Multicore Architectures". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 22.2 (Feb. 2011), pp. 309–322. DOI: 10.1109/TPDS.2010.66 (cit. on p. 103).

[Gal62] R. Gallager. "Low-Density Parity-Check Codes". In: *IRE Transactions on Information Theory* 8.1 (Jan. 1962), pp. 21–28. DOI: 10.1109/TIT.1962.1057683 (cit. on pp. 8, 12).

[GB13] S. Grönroos and J. Björkqvist. "Performance Evaluation of LDPC Decoding on a General Purpose Mobile CPU". In: *Global Conference on Signal and Information Processing (GlobalSIP)*. Dec. 2013, pp. 1278–1281. DOI: 10.1109/GlobalSIP.2013.6737142 (cit. on p. 103).

[GCC18] GCC. *Semantics of Floating Point Math in GCC*. 2018. URL: https://gcc.gnu.org/wiki/FloatingPointMath (visited on 09/16/2018) (cit. on p. 66).

[GDB10] C. Glitia, P. Dumont, and P. Boulet. "Array-OL with Delays, a Domain Specific Specification Language for Multidimensional Intensive Signal Processing". In: *Springer Multidimensional Systems and Signal Processing* 21 (Mar. 2010), pp. 105–131. DOI: 10.1007/s11045-009-0085-4 (cit. on p. 115).

[Gha+17] A. Ghaffari, M. Léonardon, Y. Savaria, C. Jégo, and C. Leroux. "Improving Performance of SCMA MPA Decoders using Estimation of Conditional Probabilities". In: *International Conference on New Circuits and Systems (NEWCAS)*. June 2017, pp. 21–24. DOI: 10.1109/NEWCAS.2017.8010095 (cit. on pp. 29, 78).

[Gha+18] R. Ghanaatian, A. Balatsoukas-Stimming, T. C. Müller, M. Meidlinger, G. Matz, A. Teman, and A. Burg. "A 588-Gb/s LDPC Decoder Based on Finite-Alphabet Message Passing". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.2 (Feb. 2018), pp. 329–340. DOI: 10.1109/TVLSI.2017.2766925 (cit. on p. 84).

[Gia+14] P. Giard, G. Sarkis, C. Thibeault, and W. J. Gross. "Fast Software Polar Decoders". In: *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, May 2014, pp. 7555–7559. DOI: 10.1109/ICASSP.2014.6855069 (cit. on pp. 47, 94–96, 104).

[Gia+16] P. Giard, G. Sarkis, C. Leroux, C. Thibeault, and W. J. Gross. "Low-Latency Software Polar Decoders". In: *Springer Journal of Signal Processing Systems (JSPS)* 90 (July 11, 2016), pp. 761–775. DOI: 10.1007/s11265-016-1157-y (cit. on pp. 32, 54, 95, 104, 115, 135).

[GNB12] S. Grönroos, K. Nybom, and J. Björkqvist. "Efficient GPU and CPU-Based LDPC Decoders for Long Codewords". In: *Springer Journal of Analog Integrated Circuits and Signal Processing (AICSP)* 73.2 (Nov. 1, 2012), p. 583. DOI: 10.1007/s10470-012-9895-7 (cit. on p. 103).

[GO19] A. Guermouche and A-C. Orgerie. *Experimental Analysis of Vectorized Instructions Impact on Energy and Power Consumption under Thermal Design Power Constraints*. Research rep. working paper or preprint. Télécom SudParis and Inria Rennes - Bretagne Atlantique, June 2019. URL: https://hal.archives-ouvertes.fr/hal-02167083v2 (cit. on p. 85).

[Gra19] E. Grayver. "Scaling the Fast x86 DVB-S2 Decoder to 1 Gbps". In: *Aerospace Conference (AeroConf)*. IEEE, Mar. 2019, pp. 1–9. DOI: 10.1109/AERO.2019.8742225 (cit. on pp. 103, 131).

[GU20] E. Grayver and A. Utter. "Extreme Software Defined Radio – GHz in Real Time". In: *Aerospace Conference (AeroConf)*. IEEE, Mar. 2020. arXiv: 2001.03645 (cit. on pp. 115, 131).

[Han+17] X. Han, R. Liu, Z. Liu, and L. Zhao. "Successive-Cancellation List Decoder of Polar Codes based on GPU". In: *International Conference on Computer and Communications (ICCC)*. IEEE, Dec. 2017, pp. 2065–2070. DOI: 10.1109/CompComm.2017.8322900 (cit. on p. 104).

[HNH13] X. Han, K. Niu, and Z. He. "Implementation of IEEE 802.11n LDPC Codes Based on General Purpose Processors". In: *International Conference on Communication Technology (ICCT)*. IEEE, Nov. 2013, pp. 218–222. DOI: 10.1109/ICCT.2013.6820375 (cit. on p. 103).

[Hoc59] A. Hocquenghem. "Codes correcteurs d'erreurs". In: *Chiffres*. Vol. 2. Paris, Sept. 1959, pp. 147–156. URL: http://kom.aau.dk/~heb/kurser/NOTER/KOFA02.PDF (cit. on p. 8).

[How15] L. Howes. *The OpenCL Specification*. Version 2.1, Revision 23. 2015. URL: https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf (cit. on p. 35).

[Hua+11] L. Huang, Y. Luo, H. Wang, F. Yang, Z. Shi, and D. Gu. "A High Speed Turbo Decoder Implementation for CPU-Based SDR System". In: *International Conference on Communication Technology and Applications (ICCTA)*. IEEE, Oct. 2011, pp. 19–23. DOI: 10.1049/cp.2011.0622 (cit. on pp. 100, 105).

[Hua13] Huawei. *5G: A Technology Vision*. Tech. rep. Huawei, 2013. URL: https://www.huawei.com/ilink/en/download/HW_314849 (cit. on p. 25).

[HV20] K. Hsieh and R. Venkataramanan. *Modulated Sparse Superposition Codes for the Complex AWGN Channel*. Apr. 2020. arXiv: 2004.09549 [cs.IT] (cit. on p. 84).

[IS18] A. Inan and J. Schiefer. *Playing with Low-Density Parity-Check Codes*. 2018. URL: https://github.com/xdsopl/LDPC (cit. on p. 72).

[Isl+17] S. M. R. Islam, N. Avazov, O. A. Dobre, and K. Kwak. "Power-Domain Non-Orthogonal Multiple Access (NOMA) in 5G Systems: Potentials and Challenges". In: *IEEE Communications Surveys Tutorials* 19.2 (2017), pp. 721–742. DOI: 10.1109/COMST.2016.2621116 (cit. on p. 26).

[JCS11] H. Ji, J. Cho, and W. Sung. "Memory Access Optimized Implementation of Cyclic and Quasi-Cyclic LDPC Codes on a GPGPU". In: *Springer Journal of Signal Processing Systems (JSPS)* 64.1 (July 1, 2011), p. 149. DOI: 10.1007/s11265-010-0547-9 (cit. on p. 103).

[Kan17] P. Kanapickas. *libsimdpp*. 2017. URL: https://github.com/p12tic/libsimdpp (cit. on p. 36).

[Kar+13] A. Karlsson, J. Sohl, J. Wang, and D. Liu. "ePUMA: A Unique Memory Access based Parallel DSP Processor for SDR and CR". In: *Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, Dec. 2013, pp. 1234–1237. DOI: 10.1109/GlobalSIP.2013.6737131 (cit. on p. 115).

[KK17a] S. Keskin and T. Kocak. "GPU Accelerated Gigabit Level BCH and LDPC Concatenated Coding System". In: *High Performance Extreme Computing Conference (HPEC)*. IEEE, Sept. 2017, pp. 1–4. DOI: 10.1109/HPEC.2017.8091021 (cit. on pp. 103, 115).

[KK17b] S. Keskin and T. Kocak. "GPU-Based Gigabit LDPC Decoder". In: *IEEE Communications Letters (COMML)* 21.8 (Aug. 2017), pp. 1703–1706. DOI: 10.1109/LCOMM.2017.2704113 (cit. on pp. 103, 135).

[KL12] M. Kretz and V. Lindenstruth. "Vc: A C++ Library for Explicit Vectorization". In: *Software: Practice and Experience* 42.11 (2012), pp. 1409–1430. DOI: 10.1002/spe.1149 (cit. on p. 36).

[KM12] S. Kang and J. Moon. "Parallel LDPC Decoder Implementation on GPU Based on Unbalanced Memory Coalescing". In: *International Conference on Communications (ICC)*. IEEE, June 2012, pp. 3692–3697. DOI: 10.1109/ICC.2012.6363991 (cit. on p. 103).

[Knu73] D.E. Knuth. *The Art of Computer Programming*. 3. Addison-Wesley, 1973, pp. 207–209 (cit. on p. 57).

[KR08] G. Kaur and V. Raj. "Multirate Digital Signal Processing for Software Defined Radio (SDR) Technology". In: *International Conference on Emerging Trends in Engineering and Technology (ICETET)*. IEEE, July 2008, pp. 110–115. DOI: 10.1109/ICETET.2008.207 (cit. on p. 115).

[KS16] V. P. Klimentyev and A. B. Sergienko. "Detection of SCMA Signal with Channel Estimation Error". In: *Conference of Open Innovations Association and Seminar on Information Security and Protection of Information Technology (FRUCT-ISPIT)*. IEEE, Apr. 2016, pp. 106–112. DOI: 10.1109/FRUCT-ISPIT.2016.7561515 (cit. on p. 78).

[KS17] V. P. Klimentyev and A. B. Sergienko. "SCMA Codebooks Optimization Based on Genetic Algorithm". In: *European Wireless Conference*. IEEE, May 2017, pp. 1–6. URL: https://ieeexplore.ieee.org/document/8011314 (cit. on p. 78).

[KSK19] Y. Krainyk, I. Sidenko, and O Kylymovych. "Software Models for Investigation of Turbo-Product-codes Decoding". In: *International Conference on ICT in Education, Research, and Industrial Applications (ICTERI)*. June 2019. URL: http://ceur-ws.org/Vol-2387/20190152.pdf (cit. on p. 85).

[Kun14] P. Kundert. *EZPWD Reed-Solomon*. 2014. URL: https://github.com/pjkundert/ezpwd-reed-solomon (cit. on p. 72).

[Kun18] D. Kun. "High Throughput GPU LDPC Encoder and Decoder for DVB-S2". In: *Aerospace Conference (AeroConf)*. IEEE, Mar. 2018, pp. 1–9. DOI: 10.1109/AERO.2018.8396831 (cit. on p. 103).

[Lai+16] B. C. C. Lai, C. Y. Lee, T. H. Chiu, H. K. Kuo, and C. K. Chang. "Unified Designs for High Performance LDPC Decoding on GPGPU". In: *IEEE Transactions on Computers (TC)* 65.12 (Dec. 2016), pp. 3754–3765. DOI: 10.1109/TC.2016.2547379 (cit. on pp. 103, 135).

[LCL18] F. Lemaitre, B. Couturier, and L. Lacassagne. "Small SIMD Matrices for CERN High Throughput Computing". In: *Workshop on Programming Models for SIMD/Vector Processing (WPMVP)*. Vienna, Austria: ACM, Feb. 2018. DOI: 10.1145/3178433.3178434 (cit. on p. 85).

[Le 15] B. Le Gal. *Fast LDPC Decoder for x86*. 2015. URL: https://github.com/blegal/Fast_LDPC_decoder_for_x86 (cit. on p. 72).

[Léo+18a] M. Léonardon, C. Leroux, D. Binet, J.M. P. Langlois, C. Jégo, and Y. Savaria. "Custom Low Power Processor for Polar Decoding". In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, May 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351739 (cit. on p. 85).

[Léo+18b] M. Léonardon, C. Leroux, P. Jääskeläinen, C. Jégo, and Y. Savaria. "Transport Triggered Polar Decoders". In: *International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*. IEEE, 2018. DOI: 10.1109/ISTC.2018.8625310 (cit. on p. 84).

[Ler+13] C. Leroux, A. J. Raymond, G. Sarkis, and W. J. Gross. "A Semi-Parallel Successive-Cancellation Decoder for Polar Codes". In: *IEEE Transactions on Signal Processing (TSP)* 61.2 (Jan. 2013), pp. 289–299. DOI: 10.1109/TSP.2012.2223693 (cit. on p. 58).

[Li+13] R. Li, J. Zhou, Y. Dou, S. Guo, D. Zou, and S. Wang. "A Multi-Standard Efficient Column-Layered LDPC Decoder for Software Defined Radio on GPUs". In: *International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*. IEEE, June 2013, pp. 724–728. DOI: 10.1109/SPAWC.2013.6612145 (cit. on p. 103).

[Li+14] R. Li, Y. Dou, J. Xu, X. Niu, and S. Ni. "An Efficient Parallel SOVA-based Turbo Decoder for Software Defined Radio on GPU". In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 97.5 (2014), pp. 1027–1036. DOI: 10.1587/transfun.E97.A.1027 (cit. on pp. 59, 105, 115).

[Li+16] A. Li, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo. "Implementation of a Fully-Parallel Turbo Decoder on a General-Purpose Graphics Processing Unit". In: *IEEE Access* 4 (2016), pp. 5624–5639. DOI: 10.1109/ACCESS.2016.2586309 (cit. on p. 105).

[Lia+06] S.-W. Liao, Z. Du, G. Wu, and G.-Y. Lueh. "Data and Computation Transformations for Brook Streaming Applications on Multiprocessors". In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, Mar. 2006, pp. 207–219. DOI: 10.1109/CGO.2006.13 (cit. on p. 115).

[Liu+13] C. Liu, Z. Bie, C. Chen, and X. Jiao. "A Parallel LTE Turbo Decoder on GPU". In: *International Conference on Communication Technology (ICCT)*. IEEE, Nov. 2013, pp. 609–614. DOI: 10.1109/ICCT.2013.6820447 (cit. on pp. 59, 105).

[LJ15] B. Le Gal and C. Jégo. "High-Throughput LDPC Decoder on Low-Power Embedded Processors". In: *IEEE Communications Letters (COMML)* 19.11 (Nov. 2015), pp. 1861–1864. DOI: 10.1109/LCOMM.2015.2477081 (cit. on p. 32).

[LJ16] B. Le Gal and C. Jégo. "High-Throughput Multi-Core LDPC Decoders Based on x86 Processor". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 27.5 (May 2016), pp. 1373–1386. DOI: 10.1109/TPDS.2015.2435787 (cit. on pp. 32, 45, 89, 103, 131).

[LJ17] B. Le Gal and C. Jégo. "Low-Latency Software LDPC Decoders for x86 Multi-Core Devices". In: *International Workshop on Signal Processing Systems (SiPS)*. IEEE, Oct. 2017, pp. 1–6. DOI: 10.1109/SiPS.2017.8110001 (cit. on pp. 89, 103, 115).

[LJ19] B. Le Gal and C. Jégo. "Low-latency and High-throughput Software Turbo Decoders on Multi-core Architectures". In: *Springer Annals of Telecommunications* 75 (Aug. 2019), pp. 27–42. DOI: 10.1007/s12243-019-00727-5 (cit. on pp. 32, 45, 85, 101, 102, 105, 115).

[LJC14] B. Le Gal, C. Jégo, and J. Crenne. "A High Throughput Efficient Approach for Decoding LDPC Codes onto GPU Devices". In: *IEEE Embedded Systems Letters (ESL)* 6.2 (June 2014), pp. 29–32. DOI: 10.1109/LES.2014.2311317 (cit. on pp. 103, 115, 135).

[LL16] Y. Li and R. Liu. "High Throughput GPU Polar Decoder". In: *International Conference on Computer and Communications (ICCC)*. IEEE, Oct. 2016, pp. 1123–1127. DOI: 10.1109/CompComm.2016.7924879 (cit. on p. 104).

[LLJ14] B. Le Gal, C. Leroux, and C. Jégo. "Software Polar Decoder on an Embedded Processor". In: *International Workshop on Signal Processing Systems (SiPS)*. IEEE, Oct. 2014, pp. 1–6. DOI: 10.1109/SiPS.2014.6986083 (cit. on pp. 94, 104).

[LLJ15] B. Le Gal, C. Leroux, and C. Jégo. "Multi-Gb/s Software Decoding of Polar Codes". In: *IEEE Transactions on Signal Processing (TSP)* 63.2 (Jan. 2015), pp. 349–359. DOI: 10.1109/TSP.2014.2371781 (cit. on pp. 48, 56, 78, 94, 97, 104, 115).

[LLJ18] B. Le Gal, C. Leroux, and C. Jégo. "High-Performance Software Implementation of SCAN Decoders for Polar codes". In: *Springer Annals of Telecommunications* 73.5 (June 2018), pp. 401–412. DOI: 10.1007/s12243-018-0634-7 (cit. on pp. 32, 104).

[Llo09] J. Lloyd. *FECpp: Erasure Codes based on Vandermonde Matrices*. 2009. URL: https://github.com/randombit/fecpp (cit. on p. 72).

[LM87] E. A. Lee and D. G. Messerschmitt. "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing". In: *IEEE Transactions on Computers (TC)* C-36.1 (Jan. 1987), pp. 24–35. DOI: 10.1109/TC.1987.5009446 (cit. on p. 115).

[LN14] Y. Lin and W. Niu. "High Throughput LDPC Decoder on GPU". In: *IEEE Communications Letters (COMML)* 18.2 (Feb. 2014), pp. 344–347. DOI: 10.1109/LCOMM.2014.010214.132406 (cit. on p. 103).

[LST12] B. Li, H. Shen, and D. Tse. "An Adaptive Successive Cancellation List Decoder for Polar Codes with Cyclic Redundancy Check". In: *IEEE Communications Letters (COMML)* 16.12 (Dec. 2012), pp. 2044–2047. DOI: 10.1109/LCOMM.2012.111612.121898 (cit. on p. 17).

[Lu+15] L. Lu, Y. Chen, W. Guo, H. Yang, Y. Wu, and S. Xing. "Prototype for 5G New Air Interface Technology SCMA and Performance Evaluation". In: *IEEE China Communications* 12.Supplement (Dec. 2015), pp. 38–48. DOI: 10.1109/CC.2015.7386169 (cit. on p. 26).

[LXY14] J. Lin, C. Xiong, and Z. Yan. "A Reduced Latency List Decoding Algorithm for Polar Codes". In: *International Workshop on Signal Processing Systems (SiPS)*. IEEE, Oct. 2014, pp. 1–6. DOI: 10.1109/SiPS.2014.6986062 (cit. on p. 50).

[Mab17] J. Mabille. *xsimd*. 2017. URL: https://github.com/xtensor-stack/xsimd (cit. on p. 36).

[MBB15] M. R. Maheshwarappa, M. Bowyer, and C. P. Bridges. "Software Defined Radio (SDR) Architecture to Support Multi-satellite Communications". In: *Aerospace Conference (AeroConf)*. IEEE, Mar. 2015, pp. 1–10. DOI: 10.1109/AERO.2015.7119186 (cit. on p. 115).

[MBJ09] O. Muller, A. Baghdadi, and M. Jezequel. "From Parallelism Levels to a Multi-ASIP Architecture for Turbo Decoding". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17.1 (Jan. 2009), pp. 92–102. DOI: 10.1109/TVLSI.2008.2003164 (cit. on pp. 59, 60).

[Mit92] J. Mitola. "Software Radios-Survey, Critical Evaluation and Future Directions". In: *NTC-92: National Telesystems Conference*. IEEE, May 1992, pp. 13/15–13/23. DOI: 10.1109/NTC.1992.267870 (cit. on p. 25).

[Mit93] J. Mitola. "Software Radios: Survey, Critical Evaluation and Future Directions". In: *IEEE Aerospace and Electronic Systems Magazine* 8.4 (Apr. 1993), pp. 25–36. DOI: 10.1109/62.210638 (cit. on pp. 25, 115).

[MK19] S. Meshram and N. Kolhare. "The Advent Software Defined Radio: FM Receiver with RTL SDR and GNU Radio". In: *International Conference on Smart Systems and Inventive Technology (ICSSIT)*. IEEE, Nov. 2019, pp. 230–235. DOI: 10.1109/ICSSIT46314.2019.8987588 (cit. on p. 115).

[MMA19] N. A. Mohammed, A. M. Mansoor, and R. B. Ahmad. "Mission-Critical Machine-Type Communication: An Overview and Perspectives Towards 5G". In: *IEEE Access* 7 (2019), pp. 127198–127216. DOI: 10.1109/ACCESS.2019.2894263 (cit. on p. 85).

[MN95] D. J. C. MacKay and R. M. Neal. "Good Codes Based on Very Sparse Matrices". In: *IMA International Conference on Cryptography and Coding (IMA-CCC)*. UK: Springer, Dec. 1995, pp. 100–111. DOI: 10.1007/3-540-60693-9_13 (cit. on pp. 8, 12).

[MN98] M. Matsumoto and T. Nishimura. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1 (1998), pp. 3–30. DOI: 10.1145/272991.272995 (cit. on p. 42).

[Möl16] R. Möller. *Design of a Low-Level C++ Template SIMD Library*. Tech. rep. Bielefeld University, Faculty of Technology, Computer Engineering Group, 2016. URL: http://www.ti.uni-bielefeld.de/html/people/moeller/tsimd_warpingsimd.html (cit. on p. 36).

[Mor06] R. H. Morelos-Zaragoza. *the-art-of-ecc.com*. 2006. URL: http://www.the-art-of-ecc.com (cit. on p. 72).

[Mor89] R. H. Morelos-Zaragoza. *The Error Correcting Codes (ECC) Page*. 1989. URL: http://www.eccpage.com (cit. on p. 72).

[Mul54] D. E. Muller. "Application of Boolean Algebra to Switching Circuit Design and to Error Detection". In: *Transactions of the IRE Professional Group on Electronic Computers* EC-3.3 (Sept. 1954), pp. 6–12. DOI: 10.1109/IREPGELC.1954.6499441 (cit. on p. 9).

[NB13] H. Nikopour and H. Baligh. "Sparse Code Multiple Access". In: *International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*. IEEE, Sept. 2013, pp. 332–336. DOI: 10.1109/PIMRC.2013.6666156 (cit. on pp. 26, 78).

[NC18] K. Natarajan and N. Chandrachoodan. “Lossless Parallel Implementation of a Turbo Decoder on GPU”. In: *International Conference on High Performance Computing (HiPC)*. Dec. 2018, pp. 133–142. DOI: 10.1109/HiPC.2018.00023 (cit. on p. 85).

[Nea06] R. Neal. *Software for Low Density Parity Check codes*. 2006. URL: https://github.com/radfordneal/LDPC-codes (cit. on p. 72).

[NGM15] NGMN Alliance. *5G White Paper*. 2015. URL: https://www.ngmn.org/wp-content/uploads/NGMN_5G_White_Paper_V1_0.pdf (cit. on p. 26).

[Nik15] N. Nikaein. “Processing Radio Access Network Functions in the Cloud: Critical Issues and Modeling”. In: *International Workshop on Mobile Cloud Computing and Services (MCS)*. ACM, 2015, pp. 36–43. DOI: 10.1145/2802130.2802136 (cit. on p. 26).

[NRV16] R. Nivin, J. S. Rani, and P. Vidhya. “Design and Hardware Implementation of Reconfigurable Nano Satellite Communication System using FPGA based SDR for FM/FSK Demodulation and BPSK Modulation”. In: *International Conference on Communication Systems and Networks (ComNet)*. IEEE, July 2016, pp. 1–6. DOI: 10.1109/CSN.2016.7823976 (cit. on p. 115).

[Ope13] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. 2013. URL: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf (cit. on p. 35).

[pab16] pabr. *leansdr: Lightweight, Portable Software-defined Radio*. 2016. URL: https://github.com/pabr/leansdr (cit. on p. 131).

[Pal+10] M. Palkovic, P. Raghavan, M. Li, A. Dejonghe, L. Van der Perre, and F. Catthoor. “Future Software-Defined Radio Platforms and Mapping Flows”. In: *IEEE Signal Processing Magazine* 27.2 (Mar. 2010), pp. 22–33. DOI: 10.1109/MSP.2009.935386 (cit. on p. 115).

[Pal+12] M. Palkovic, J. Declerck, P. Avasare, M. Glassee, A. Dewilde, P. Raghavan, A. Dejonghe, and L. Van der Perre. “DART - a High Level Software-Defined Radio Platform Model for Developing the Run-Time Controller”. In: *Springer Journal of Signal Processing Systems (JSPS)* 69 (Mar. 2012), pp. 317–327. DOI: 10.1007/s11265-012-0669-3 (cit. on p. 115).

[Pan+13] Xia Pan, Xiao-fan Lu, Ming-qi Li, and Rong-fang Song. “A High Throughput LDPC Decoder in CMMB Based on Virtual Radio”. In: *Wireless Communications and Networking Conference Workshops (WCNCW)*. IEEE, Apr. 2013, pp. 95–99. DOI: 10.1109/WCNCW.2013.6533323 (cit. on p. 103).

[Par10] A. Partow. *Schifra: Reed-Solomon Error Correcting Code Library for Software Applications Implemented in C++*. 2010. URL: https://github.com/ArashPartow/schifra (cit. on p. 72).

[Pig+18] V. Pignoly, B. Le Gal, C. Jégo, and B. Gadat. “High Data Rate and Flexible Hardware QC-LDPC Decoder for Satellite Optical Communications”. In: *International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*. IEEE, Dec. 2018, pp. 1–5. DOI: 10.1109/ISTC.2018.8625274 (cit. on p. 84).

[PM12] M. Pharr and W. R. Mark. “ispc: A SPMD Compiler for High-Performance CPU Programming”. In: *Innovative Parallel Computing (InPar)*. IEEE, May 2012, pp. 1–13. DOI: 10.1109/InPar.2012.6339601 (cit. on p. 35).

[Poh+16] A. Pohl, B. Cosenza, M. A. Mesa, C. C. Chi, and B. Juurlink. "An Evaluation of Current SIMD Programming Models for C++". In: *Workshop on Programming Models for SIMD/Vector Processing (WPMVP)*. Barcelona, Spain: ACM, 2016, 3:1–3:8. DOI: 10.1145/2870650.2870653 (cit. on p. 35).

[Pou+18] S. Poulenard, B. Gadat, J. F. Chouteau, T. Anfray, C. Poulliat, C. Jego, O. Hartmann, G. Artaud, and H. Meric. "Forward Error Correcting Code for High Data Rate LEO Satellite Optical Downlinks". In: *International Conference on Space Optics (ICSO)*. Ed. by Zoran Sodnik, Nikos Karafolas, and Bruno Cugny. Vol. 11180. International Society for Optics and Photonics. SPIE, 2018, pp. 2029–2038. DOI: 10.1117/12.2536120 (cit. on p. 84).

[PPL95] T. M. Parks, J. L. Pino, and E. A. Lee. "A Comparison of Synchronous and Cycle-Static Dataflow". In: *Asilomar Conference on Signals, Systems, and Computers (ACSSC)*. Vol. 1. IEEE, Oct. 1995, pp. 204–210. DOI: 10.1109/ACSSC.1995.540541 (cit. on p. 115).

[PT+15] T. T. Pham, T. Tsou, et al. *TurboFEC - SIMD Vectorized LTE Turbo and Convolutional Encoders and Decoders*. 2015. URL: https://github.com/ttsou/turbofec (cit. on p. 72).

[Pyn+94] R. Pyndiah, A. Glavieux, A. Picart, and S. Jacq. "Near Optimum Decoding of Product Codes". In: *Global Communications Conference (GLOBECOM)*. Vol. 1. IEEE, Nov. 1994, pp. 339–343. DOI: 10.1109/GLOCOM.1994.513494 (cit. on p. 8).

[Ree54] I. Reed. "A Class of Multiple-error-correcting Codes and the Decoding Scheme". In: *Transactions of the IRE Professional Group on Information Theory* 4.4 (Sept. 1954), pp. 38–49. DOI: 10.1109/TIT.1954.1057465 (cit. on p. 9).

[RG17] V. Q. Rodriguez and F. Guillemin. "Towards the Deployment of a Fully Centralized Cloud-RAN Architecture". In: *International Wireless Communications and Mobile Computing Conference (IWCMC)*. IEEE, June 2017, pp. 1055–1060. DOI: 10.1109/IWCMC.2017.7986431 (cit. on p. 26).

[RHV20] C. Rush, K. Hsieh, and R. Venkataramanan. *Capacity-achieving Spatially Coupled Sparse Superposition Codes with AMP Decoding*. Feb. 2020. arXiv: 2002.07844 [cs.IT] (cit. on p. 84).

[RL09] W. Ryan and S. Lin. *Channel Codes: Classical and Modern*. Cambridge University Press, Sept. 2009. DOI: 10.1017/CBO9780511803253 (cit. on p. 131).

[Rob13] A. D. Robison. "Composable Parallel Patterns with Intel Cilk Plus". In: *IEEE Computing in Science & Engineering (CS&E)* 15.2 (Mar. 2013), pp. 66–71. DOI: 10.1109/MCSE.2013.21 (cit. on p. 35).

[Ron+06] T. Rondeau, J. Blum, J. Corgan, S. Koslowski, E. Blossom, M. Müller, T. O'Shea, B. Reynwar, M. Dickens, A. Rode, R. Economos, M. Braun, et al. *GNURadio: the Free and Open Software Radio Ecosystem*. 2006. URL: https://github.com/gnuradio/gnuradio (cit. on pp. 72, 116).

[Ros+14] P. Rost, C. J. Bernardos, A. D. Domenico, M. D. Girolamo, M. Lalam, A. Maeder, D. Sabella, and D. Wübben. "Cloud Technologies for Flexible 5G Radio Access Networks". In: *IEEE Communications Magazine* 52.5 (May 2014), pp. 68–76. DOI: 10.1109/MCOM.2014.6898939 (cit. on pp. 25, 115).

[RS60] I. Reed and G. Solomon. "Polynomial Codes Over Certain Finite Fields". In: *Journal of the Society for Industrial and Applied Mathematics* 8.2 (1960), pp. 300–304. DOI: 10.1137/0108018 (cit. on p. 8).

[SA13] S. Shaik and S. Angadi. "Architecture and Component Selection for SDR Applications". In: *International Journal of Engineering Trends and Technology (IJETT)* 4.4 (2013), pp. 691–694. URL: http://www.ijettjournal.org/volume-4/issue-4/IJETT-V4I4P236.pdf (cit. on p. 115).

[Sar+14a] G. Sarkis, P. Giard, C. Thibeault, and W. J. Gross. "Autogenerating Software Polar Decoders". In: *Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, Dec. 2014, pp. 6–10. DOI: 10.1109/GlobalSIP.2014.7032067 (cit. on pp. 94, 96, 97, 99, 104).

[Sar+14b] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross. "Fast Polar Decoders: Algorithm and Implementation". In: *IEEE Journal on Selected Areas in Communications (JSAC)* 32.5 (May 2014), pp. 946–957. DOI: 10.1109/JSAC.2014.140514 (cit. on pp. 16, 94).

[Sar+14c] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross. "Increasing the Speed of Polar List Decoders". In: *International Workshop on Signal Processing Systems (SiPS)*. IEEE, Oct. 2014, pp. 1–6. DOI: 10.1109/SiPS.2014.6986089 (cit. on pp. 57, 58, 99, 104).

[Sar+16] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross. "Fast List Decoders for Polar Codes". In: *IEEE Journal on Selected Areas in Communications (JSAC)* 34.2 (Feb. 2016), pp. 318–328. DOI: 10.1109/JSAC.2015.2504299 (cit. on pp. 16, 32, 50, 56–58, 98, 99, 104).

[SBW06] K. Skey, J. Bradley, and K. Wagner. "A Reuse Approach for FPGA-Based SDR Waveforms". In: *Military Communications Conference (MILCOM)*. IEEE, Oct. 2006, pp. 1–7. DOI: 10.1109/MILCOM.2006.302391 (cit. on p. 115).

[Sch32] J. Schreier. "On Tournament Elimination Systems". In: *Mathesis Polska* 7 (1932), pp. 154–160. URL: https://ci.nii.ac.jp/naid/10027928626/en/ (cit. on p. 57).

[Sha+19] F. Shaheen, M. F. U. Butt, S. Agha, S. X. Ng, and R. G. Maunder. "Performance Analysis of High Throughput MAP Decoder for Turbo Codes and Self Concatenated Convolutional Codes". In: *IEEE Access* 7 (2019), pp. 138079–138093. DOI: 10.1109/ACCESS.2019.2942152 (cit. on p. 85).

[Sha48] C. E. Shannon. "A Mathematical Theory of Communication". In: *The Bell System Technical Journal* 27.4 (Oct. 1948), pp. 623–656. DOI: 10.1002/j.1538-7305.1948.tb00917.x (cit. on pp. 6, 7).

[She+16] Y. Shen, C. Zhang, J. Yang, S. Zhang, and X. You. "Low-Latency Software Successive Cancellation List Polar Decoder using Stage-Located Copy". In: *International Conference on Digital Signal Processing (DSP)*. IEEE, Oct. 2016, pp. 84–88. DOI: 10.1109/ICDSP.2016.7868521 (cit. on pp. 57, 58, 99, 104).

[She+20] Y. Shen, L. Li, J. Yang, X. Tan, Z. Zhang, X. You, and C. Zhang. "Low-Latency Segmented List-Pruning Software Polar List Decoder". In: *IEEE Transactions on Vehicular Technology* 69.4 (Apr. 2020), pp. 3575–3589. DOI: 10.1109/TVT.2020.2973552 (cit. on p. 85).

[Sho04] A. Shokrollahi. "Raptor Codes". In: *International Symposium on Information Theory (ISIT)*. IEEE, June 2004, pp. 36–. DOI: 10.1109/ISIT.2004.1365073 (cit. on p. 9).

[Str13] Bjarne Stroustrup. *The C++ Programming Language*. 4th. Addison-Wesley Professional, 2013 (cit. on pp. 51, 70).

[Str20] B. Stroustrup. "Thriving in a Crowded and Changing World: C++ 2006–2020". In: *Proceedings of the ACM on Programming Languages (PACMPL)* 4.HOPL (June 2020). DOI: 10.1145/3386320 (cit. on p. 70).

[Stu+11] C. Studer, C. Benkeser, S. Belfanti, and Q. Huang. "Design and Implementation of a Parallel Turbo-Decoder ASIC for 3GPP-LTE". In: *IEEE Journal of Solid-State Circuits (JSSC)* 46.1 (Jan. 2011), pp. 8–17. DOI: 10.1109/JSSC.2010.2075390 (cit. on p. 22).

[SWC17] G. Song, X. Wang, and J. Cheng. "Signature Design of Sparsely Spread Code Division Multiple Access Based on Superposed Constellation Distance Analysis". In: *IEEE Access* 5 (Oct. 2017), pp. 23809–23821. DOI: 10.1109/ACCESS.2017.2765346 (cit. on p. 78).

[TA10] W. Thies and S. Amarasinghe. "An Empirical Characterization of Stream Programs and its Implications for Language and Compiler Design". In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM/IEEE, Sept. 2010, pp. 365–376. DOI: 10.1145/1854273.1854319 (cit. on p. 115).

[Tav16a] S. Tavildar. *C and MATLAB Implementations for LDPC Codes*. 2016. URL: https://github.com/tavildar/LDPC (cit. on p. 72).

[Tav16b] S. Tavildar. *C and MATLAB Implementations for Polar Codes*. 2016. URL: https://github.com/tavildar/Polar (cit. on p. 72).

[Tay+17] C. Taylor, E. Nemerson, M. Corallo, and M. Al-Bassam. *Leopard-RS: MDS Reed-Solomon Erasure Correction Codes for Large Data in C*. 2017. URL: https://github.com/catid/leopard (cit. on p. 72).

[Tay18a] C. Taylor. *FEC-AL: Forward Error Correction at the Application Layer in C*. 2018. URL: https://github.com/catid/fecal (cit. on p. 72).

[Tay18b] C. Taylor. *Siamese: Infinite-Window Streaming Erasure Code (HARQ)*. 2018. URL: https://github.com/catid/siamese (cit. on p. 72).

[TB20] A. Tasdighi and E. Boutillon. *Integer Ring Sieve (IRS) for Constructing Compact QC-LDPC Codes with Large Girth*. Mar. 2020. arXiv: 2003.08707 [cs.IT] (cit. on p. 84).

[Ton+16a] T. Tonnellier, C. Leroux, B. Le Gal, B. Gadat, C. Jégo, and N. Van Wambeke. "Lowering the Error Floor of Turbo Codes With CRC Verification". In: *IEEE Wireless Communications Letters (WCL)* 5.4 (Aug. 2016), pp. 404–407. DOI: 10.1109/LWC.2016.2571283 (cit. on p. 77).

[Ton+16b] T. Tonnellier, C. Leroux, B. Le Gal, C. Jégo, B. Gadat, and N. Van Wambeke. "Lowering the Error Floor of Double-Binary Turbo Codes: The Flip and Check Algorithm". In: *International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*. IEEE, Sept. 2016, pp. 156–160. DOI: 10.1109/ISTC.2016.7593096 (cit. on p. 77).

[Ton17] T. Tonnellier. "Contribution to the Improvement of the Decoding Performance of Turbo Codes : Algorithms and Architecture". PhD thesis. Université de Bordeaux, 2017. URL: https://tel.archives-ouvertes.fr/tel-01580476 (cit. on p. 77).

[Tri12] P. Trifonov. "Efficient Design and Decoding of Polar Codes". In: *IEEE Transactions on Communications (TCOM)* 60.11 (Nov. 2012), pp. 3221–3227. DOI: 10.1109/TCOMM.2012.081512.110872 (cit. on pp. 12, 15).

[TV11] I. Tal and A. Vardy. "List Decoding of Polar Codes". In: *International Symposium on Information Theory (ISIT)*. IEEE, July 2011, pp. 1–5. DOI: 10.1109/ISIT.2011.6033904 (cit. on pp. 15, 57, 58).

[Vam+19] M. Vameghestahbanati, I. D. Marsland, R. H. Gohary, and H. Yanikomeroglu. "Multidimensional Constellations for Uplink SCMA Systems – A Comparative Study". In: *IEEE Communications Surveys Tutorials* 21.3 (2019), pp. 2169–2194. DOI: 10.1109/COMST.2019.2910569 (cit. on p. 85).

[VF00] J. Vogt and A. Finger. "Improving the max-log-MAP Turbo Decoder". In: *IET Electronics Letters* 36.23 (Nov. 2000), pp. 1937–1939. DOI: 10.1049/el:20001357 (cit. on p. 22).

[Wan+11a] G. Wang, M. Wu, Y. Sun, and J. R. Cavallaro. "A Massively Parallel Implementation of QC-LDPC Decoder on GPU". In: *Symposium on Application Specific Processors (SASP)*. IEEE, June 2011, pp. 82–85. DOI: 10.1109/SASP.2011.5941084 (cit. on p. 103).

[Wan+11b] G. Wang, M. Wu, Y. Sun, and J. R. Cavallaro. "GPU Accelerated Scalable Parallel Decoding of LDPC Codes". In: *Asilomar Conference on Signals, Systems, and Computers (ACSSC)*. IEEE, Nov. 2011, pp. 2053–2057. DOI: 10.1109/ACSSC.2011.6190388 (cit. on p. 103).

[Wan+13] G. Wang, M. Wu, B. Yin, and J. R. Cavallaro. "High Throughput Low Latency LDPC Decoding on GPU for SDR Systems". In: *Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, Dec. 2013, pp. 1258–1261. DOI: 10.1109/GlobalSIP.2013.6737137 (cit. on p. 103).

[Wan+19] Y. Wang, L. Chen, Q. Wang, Y. Zhang, and Z. Xing. "Algorithm and Architecture for Path Metric Aided Bit-Flipping Decoding of Polar Codes". In: *Wireless Communications and Networking Conference (WCNC)*. Apr. 2019, pp. 1–6. DOI: 10.1109/WCNC.2019.8885419 (cit. on p. 84).

[WCW08] S. Wang, S. Cheng, and Q. Wu. "A Parallel Decoding Algorithm of LDPC Codes Using CUDA". In: *Asilomar Conference on Signals, Systems, and Computers (ACSSC)*. IEEE, Oct. 2008, pp. 171–175. DOI: 10.1109/ACSSC.2008.5074385 (cit. on p. 103).

[WSC10] M. Wu, Y. Sun, and J. R. Cavallaro. "Implementation of a 3GPP LTE Turbo Decoder Accelerator on GPU". In: *International Workshop on Signal Processing Systems (SiPS)*. IEEE, Oct. 2010, pp. 192–197. DOI: 10.1109/SIPS.2010.5624788 (cit. on pp. 59, 60, 105).

[Wu+11] M. Wu, Y. Sun, G. Wang, and J. R. Cavallaro. "Implementation of a High Throughput 3GPP Turbo Decoder on GPU". In: *Springer Journal of Signal Processing Systems (JSPS)* 65.2 (Sept. 10, 2011), p. 171. DOI: 10.1007/s11265-011-0617-7 (cit. on pp. 59, 60, 105, 135).

[Wu+13] M. Wu, G. Wang, B. Yin, C. Studer, and J. R. Cavallaro. "HSPA+/LTE-A Turbo Decoder on GPU and Multicore CPU". In: *Asilomar Conference on Signals, Systems, and Computers (ACSSC)*. IEEE, Nov. 2013, pp. 824–828. DOI: 10.1109/ACSSC.2013.6810402 (cit. on pp. 32, 59, 60, 100–102, 105).

[Wub+14] D. Wubben, P. Rost, J. S. Bartelt, M. Lalam, V. Savin, M. Gorgoglione, A. Dekorsy, and G. Fettweis. "Benefits and Impact of Cloud Computing on 5G Signal Processing: Flexible Centralization Through Cloud-RAN". In: *IEEE Signal Processing Magazine* 31.6 (Nov. 2014), pp. 35–44. DOI: 10.1109/MSP.2014.2334952 (cit. on p. 25).

[WZC15] Y. Wu, S. Zhang, and Y. Chen. "Iterative Multiuser Receiver in Sparse Code Multiple Access Systems". In: *International Conference on Communications (ICC)*. IEEE, June 2015, pp. 2918–2923. DOI: 10.1109/ICC.2015.7248770 (cit. on p. 78).

[Xia+13] J. Xianjun, C. Canfeng, P. Jääskeläinen, V. Guzma, and H. Berg. "A 122Mb/s Turbo decoder using a mid-range GPU". In: *International Wireless Communications and Mobile Computing Conference (IWCMC)*. IEEE, July 2013, pp. 1090–1094. DOI: 10.1109/IWCMC.2013.6583709 (cit. on pp. 59, 105, 115, 135).

[Xu+19] Y. Xu, W. Wang, Z. Xu, and X. Gao. "AVX-512 Based Software Decoding for 5G LDPC Codes". In: *International Workshop on Signal Processing Systems (SiPS)*. IEEE, Oct. 2019, pp. 54–59. DOI: 10.1109/SiPS47522.2019.9020587 (cit. on pp. 45, 89, 103).

[YC12] D. R. N. Yoge and N. Chandrachoodan. "GPU Implementation of a Programmable Turbo Decoder for Software Defined Radio Applications". In: *International Conference on VLSI Design*. IEEE, Jan. 2012, pp. 149–154. DOI: 10.1109/VLSID.2012.62 (cit. on pp. 59, 105, 115).

[Yeo+01] E. Yeo, P. Pakzad, B. Nikolic, and V. Anantharam. "High Throughput Low-Density Parity-Check Decoder Architectures". In: *Global Communications Conference (GLOBECOM)*. Vol. 5. IEEE, 2001, pp. 3019–3024. DOI: 10.1109/GLOCOM.2001.965981 (cit. on p. 12).

[Yin+12] Y. Ying, K. You, L. Zhou, H. Quan, M. Jing, Z. Yu, and X. Zeng. "A Pure Software LDPC Decoder on a Multi-Core Processor Platform with Reduced Inter-Processor Communication Cost". In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, May 2012, pp. 2609–2612. DOI: 10.1109/ISCAS.2012.6271839 (cit. on p. 106).

[Zen+17] J. Zeng, C. Wu, Z. Zhang, X. Cheng, G. Xie, J. Han, X. Zeng, and Z. Yu. "A Multi-core-based Heterogeneous Parallel Turbo Decoder". In: *IEICE Electronics Express* advpub (2017). DOI: 10.1587/elex.14.20170768 (cit. on p. 85).

[ZF02] J. Zhang and M. Fossorier. "Shuffled Belief Propagation Decoding". In: *Asilomar Conference on Signals, Systems, and Computers (ACSSC)*. Vol. 1. IEEE, Nov. 2002, pp. 8–15. DOI: 10.1109/ACSSC.2002.1197141 (cit. on p. 12).

[Zha+11] J. Zhao, M. Zhao, H. Yang, J. Chen, X. Chen, and J. Wang. "High Performance LDPC Decoder on Cell BE for WiMAX System". In: *International Conference on Communications and Mobile Computing (CMC)*. IEEE, Apr. 2011, pp. 278–281. DOI: 10.1109/CMC.2011.117 (cit. on p. 103).

[Zha+12] Suiping Zhang, Rongrong Qian, Tao Peng, Ran Duan, and Kuilin Chen. "High Throughput Turbo Decoder Design for GPP Platform". In: *International Conference on Communications and Networking in China (CHINACOM)*. IEEE, Aug. 2012, pp. 817–821. DOI: 10.1109/ChinaCom.2012.6417597 (cit. on pp. 32, 59, 60, 100–102, 105).

[Zha+14a] S. Zhang, X. Xu, L. Lu, Y. Wu, G. He, and Y. Chen. "Sparse Code Multiple Access: An Energy Efficient Uplink Approach for 5G Wireless Systems". In: *Global Communications Conference (GLOBECOM)*. IEEE, Dec. 2014, pp. 4782–4787. DOI: 10.1109/GLOCOM.2014.7037563 (cit. on p. 27).

[Zha+14b] Y. Zhang, Z. Xing, L. Yuan, C. Liu, and Q. Wang. "The Acceleration of Turbo Decoder on the Newest GPGPU of Kepler Architecture". In: *International Symposium on Communications and Information Technologies (ISCIT)*. IEEE, Sept. 2014, pp. 199–203. DOI: 10.1109/ISCIT.2014.7011900 (cit. on pp. 59, 105).

[Zha+16] S. Zhang, K. Xiao, B. Xiao, Z. Chen, B. Xia, D. Chen, and S. Ma. "A Capacity-based Codebook Design Method for Sparse Code Multiple Access Systems". In: *International Conference on Wireless Communications Signal Processing (WCSP)*. IEEE, Oct. 2016, pp. 1–5. DOI: 10.1109/WCSP.2016.7752620 (cit. on p. 78).

[Zig15] B. Ziganshin. *FastECC*. 2015. URL: https://github.com/Bulat-Ziganshin/FastECC (cit. on p. 72).

Personal Publications

International Journals

[1] A. Cassagne, O. Hartmann, M. Léonardon, K. He, C. Leroux, R. Tajan, O. Aumage, D. Barthou, T. Tonnellier, V. Pignoly, B. Le Gal, and C. Jégo. "AFF3CT: A Fast Forward Error Correction Toolbox!" In: *Elsevier SoftwareX* 10 (Oct. 2019), p. 100345. DOI: 10.1016/j.softx.2019.100345 (cit. on pp. vii, ix, 72, 85, 113, 115).

[2] A. Ghaffari, M. Léonardon, A. Cassagne, C. Leroux, and Y. Savaria. "Toward High Performance Implementation of 5G SCMA Algorithms". In: *IEEE Access* 7 (2019), pp. 10402–10414. DOI: 10.1109/ACCESS.2019.2891597 (cit. on pp. vii, ix, 29, 67, 78, 108, 113, 115).

[3] M. Léonardon, A. Cassagne, C. Leroux, C. Jégo, L-P. Hamelin, and Y. Savaria. "Fast and Flexible Software Polar List Decoders". In: *Springer Journal of Signal Processing Systems (JSPS)* 91 (Jan. 2019), pp. 937–952. DOI: 10.1007/s11265-018-1430-3 (cit. on pp. vii, ix, 32, 67, 77, 99, 104, 113, 115).

International Conferences

[4] A. Cassagne, B. Le Gal, C. Leroux, O. Aumage, and D. Barthou. "An Efficient, Portable and Generic Library for Successive Cancellation Decoding of Polar Codes". In: *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Springer, Nov. 1, 2015. DOI: 10.1007/978-3-319-29778-1_19 (cit. on pp. vii, ix, 32, 51, 67, 78, 94, 96, 104, 113, 115).

[5] A. Cassagne, O. Aumage, C. Leroux, D. Barthou, and B. Le Gal. "Energy Consumption Analysis of Software Polar Decoders on Low Power Processors". In: *European Signal Processing Conference (EUSIPCO)*. IEEE, Aug. 2016, pp. 642–646. DOI: 10.1109/EUSIPCO.2016.7760327 (cit. on pp. vii, ix, 32, 51, 67, 78, 96, 104, 113, 115).

[6] A. Cassagne, T. Tonnellier, C. Leroux, B. Le Gal, O. Aumage, and D. Barthou. "Beyond Gbps Turbo decoder on multi-core CPUs". In: *International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*. IEEE, Sept. 2016, pp. 136–140. DOI: 10.1109/ISTC.2016.7593092 (cit. on pp. vii, ix, 32, 67, 78, 101, 105, 113, 115).

[7] A. Cassagne, O. Aumage, D. Barthou, C. Leroux, and C. Jégo. "MIPP: A Portable C++ SIMD Wrapper and its use for Error Correction Coding in 5G Standard". In: *Workshop on Programming Models for SIMD/Vector Processing (WPMVP)*. Vösendorf/Wien, Austria: ACM, Feb. 2018. DOI: 10.1145/3178433.3178435 (cit. on pp. vi, 35, 36, 43, 67, 115).

National Conferences and Posters

[8] A. Cassagne, O. Hartmann, M. Léonardon, T. Tonnellier, G. Delbergue, C. Leroux, R. Tajan, B. Le Gal, C. Jégo, O. Aumage, and D. Barthou. "Fast Simulation and Prototyping with AFF3CT". In: *International Workshop on Signal Processing Systems (SiPS)*. IEEE, Oct. 2017. DOI: 10.13140/RG.2.2.10295.42409/1. URL: https://sips2017.sciencesconf.org/data/demo_9.pdf (cit. on pp. vii, 76, 85, 115).

[9] A. Cassagne, M. Léonardon, O. Hartmann, T. Tonnellier, G. Delbergue, V. Giraud, C. Leroux, R. Tajan, B. Le Gal, C. Jégo, O. Aumage, and D. Barthou. "AFF3CT : Un environnement de simulation pour le codage de canal". In: *GdR SoC2*. June 2017. DOI: 10.13140/RG.2.2.13492.91520 (cit. on pp. vii, 79, 85).