UNIVERSITÉ DE MONTRÉAL

FLEXIBLE SCHEDULING OF TURBO DECODING ON A MULTIPROCESSOR PLATFORM

NEGIN SAHRAII
DEPARTEMENT DE GENIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
April 2009

Your file Votre référence
ISBN: 978-0-494-53925-5
Our file Notre référence
ISBN: 978-0-494-53925-5

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

FLEXIBLE SCHEDULING OF TURBO DECODING ON A MULTIPROCESSOR PLATFORM

présenté par : SAHRAII Negin

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

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

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

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

M. THIBEAULT Claude, Ph. D., membre et codirecteur

M. LANGLOIS J.M. Pierre, Ph. D., membre

ACKNOWLEDMENTS

I would like to thank my supervisor, Professor Yvon Savaria and my co supervisor, Professor Claude Thibeault, for their precious time and effort, invaluable guidance and intellectual support during my graduate study. I have been extremely lucky to have supervisors who cared so much about my work and encouraged me to pursue my research objectives.

I must express also my gratitude to Hojat, my husband, for his continued support and encouragement. Also, completing this work would have been difficult were it not for the moral support provided by my mother, sister, and brother. Finally, I would like to thank my friends Parissa and Sandrine for their friendships and sharing their research experience.

RÉSUMÉ

Ce projet présente un modèle de performance dynamique d'une application de communication à accès multiple et division de code à large bande (WCDMA – « wide-band code division multiple access ») programmée sur un réseau sur puce à processeurs multiples (MPSoC – « multiple processor system on chip »). Nous développons une stratégie de modélisation dynamique pour évaluer le temps d'exécution des designs MPSoC basée sur des modèles à haut niveau des applications et de l'architecture. De tels modèles permettent de s'assurer que la plate-forme multi-noyaux est exploitée à son maximum et que les stratégies d'assignation et d'ordonnancement peuvent être validées.

Nous nous sommes concentrés sur le décodeur Turbo, qui est une partie de cette application comportant un nombre important des calculs et qui présente une variabilité de temps de traitement significative. Dans un système temps-réel, il est très important que les tâches respectent leurs délais limites. En raison de la variabilité du temps d'exécution des tâches, la plupart des algorithmes d'ordonnancement utilisés dans les systèmes temps-réel sont basés sur le pire cas du temps d'exécution de l'application choisie (WCET – « worst-case execution time »). Le problème d'une méthode de conception basée sur le WCET est le suivant : l'analyse d'ordonnancement basée sur les WCET mène à un faible taux d'utilisation des processeurs. Dans le cadre de ce projet, nous proposons quelques méthodes d'ordonnancement flexibles appliquées au décodage Turbo qui sont très avantageuses en comparaison de la méthode d'ordonnancement du WCET. Les méthodes proposées sont inspirées des méthodes d'ordonnancement qui traitent de calculs flexibles. Un modèle de performance de cette application nous a permis d'implémenter et valider quelques méthodes d'ordonnancement plus flexibles proposées pour l'exécution du décodage Turbo et qui sont adaptées à l'effort de traitement variable exigé par le décodeur.

Basée le modèle de performance proposé, l'efficacité des méthodes d'ordonnancement est démontrée. Elle justifie également l'utilisation de notre modèle

d'évaluation de la performance. Les méthodes d'ordonnancement flexibles (FS – « flexible scheduling ») proposées améliorent substantiellement l'utilisation des ressources lorsque comparée à une méthode d'ordonnancement du temps d'exécution selon le pire cas (WCET). La méthode d'ordonnancement « priority-driven gradual », en comparaison de la méthode WCET, permet d'augmenter le nombre d'utilisateurs de 14 à 35, tout en maintenant une qualité de service acceptable, reflétée dans une dégradation très petite de 0.1 dB du gain de décodage.

ABSTRACT

This project presents a dynamic performance model of a Wide band Code Division Multiple Access (WCDMA) application mapped on a homogeneous Multi Processor System-on-Chip (MPSoC). We develop a dynamic modeling strategy to evaluate the performance of MPSoC designs based on high level models of the applications and of the architecture. Such model permits ensuring that the multi-core platform is well exploited.

We focus on the Turbo decoder, which is a computationally intensive part of the application and which presents significant processing time variability. In a real-time system, it is very important that the tasks meet their deadlines. Due to the variability of tasks execution time, most scheduling algorithms used in real-time systems are based on the Worst Case Execution Time (WCET) of application tasks. The problem of such WCET based design is that, the scheduling analysis based on WCETs leads to low processor utilization. In this project, some flexible scheduling methods are proposed for Turbo decoding tasks which are highly advantageous comparing to the WCET scheduling method. The proposed methods are inspired from the scheduling methods which deal with flexible computations. A performance model of this application allows deriving and validating some proposed flexible scheduling methods for Turbo decoding tasks, which are adapted to the variable processing effort required by the decoder.

Using the proposed performance model, the efficiency of the scheduling methods is demonstrated. It also justifies the utilization of our performance evaluation model. The proposed flexible scheduling (FS) methods improve the resource utilization compared to a Worst Case Execution Time (WCET) scheduling method. In a specific benchmark reported in this thesis, the priority-driven gradual scheduling method, which is the most efficient FS method among the proposed FS methods, allows increasing the number of users from 14 to 35, while keeping an acceptable

quality of service, as reflected in a very small degradation of 0.1 dB in the decoding gain.

CONDENSÉ EN FRANÇAIS

0.1. Introduction

Les plates-formes de réseau sur puces à processeurs multiples (MPSoC – « multiple processor system on chip ») peuvent fournir une puissance de traitement élevée par le partage de la charge sur un réseau des processeurs. De telles architectures sont des cibles appropriées pour l'exécution d'applications dynamiques qui nécessitent une quantité considérable de calculs. Les processeurs de traitement de signal numérique multi-cœur (DSPs – « Digital Signal Processors ») de haute performance sont de plus en plus employés dans les équipements de télécommunication qui traitent les signaux sonores, visuels et radio. Puisque ces DSP sont contrôlés par logiciel, lorsque suffisamment puissants, ils fournissent plus de flexibilité qu'un circuit intégré dédié (Application Specific Integrated Circuit – ASIC). Ces considérations ont mené au choix d'une plate-forme multi-DSP comme l'architecture cible dans ce projet. Celle-ci s'appelle Vocallo et elle a été conçue par la compagnie Octasic Semiconductor.

L'application cible de ce projet correspond au processus d'accès multiple à division de code à large bande (WCDMA – « wide-band code division multiple access ») d'une station de base de type « Universel Mobile Telecommunication Systems (UMTS) ». Le WCDMA rend possible le partage par différents utilisateurs d'une bande spectrale relativement large en l'étalement spectral par codage au lieu de tranches de temps exclusives. L'étalement de spectre (« spread spectrum ») est effectué en multipliant les flots (« stream ») de données de taux inférieur avec une séquence de taux plus élevé (connue sous le nom de « chip sequence ». Dans ce projet, nous avons l'intention de mettre en application le récepteur d'une station de base UMTS qui exploite le WCDMA sur une plate-forme multi-DSP .

Étant donné les caractéristiques temps réel et dynamiques de l'application cible et également la complexité de la conception de MPSoC, il est nécessaire d'avoir des

moyens de valider les capacités (par exemple, le trafic qu'elle peut servir) et la performance de la plate-forme avant l'implémentation, permettant ainsi de vérifier si une architecture MPSoC donnée convient à une application ou de déterminer le nombre de processeurs requis pour atteindre la performance désirée. Pour ces raisons, nous avons développé une stratégie de modélisation dynamique d'évaluation de la performance des MPSoC basée sur des modèles à niveau élevé des applications et de l'architecture. À ce niveau de modélisation, l'application est représentée comme un ensemble de tâches devant être exécutées, avec les ressources disponibles. L'architecture sur laquelle l'application s'exécutera est représentée simplement comme un ensemble de ressources de traitement reliées par un tissu de communication pour transférer des données entre elles. Un aspect exceptionnel de ce modèle est que la fonctionnalité est complètement absente du modèle, ce qui accélère la caractérisation et la conception du système. Une telle modélisation d'exécution permet une validation rapide de l'efficacité des stratégies d'assignation et d'ordonnancement d'une application complexe sur la plate-forme cible avant son exécution. La stratégie de modélisation développée est largement applicable et elle n'est pas limitée à l'application présentée dans ce projet.

Nous proposons ensuite une méthode pour l'assignation des différentes parties de l'application WCDMA sur la plate-forme. Ainsi, nous nous concentrons spécifiquement sur le processus de décodage Turbo qui est une partie de l'application WCDMA demandant une grande puissance de calcul. L'ordonnancement de ce genre de processus est étudié en détail. Le processus de décodage Turbo est caractérisé par une variabilité significative de l'effort de traitement qui rend l'ordonnancement d'un tel processus critique. Il est bien connu que les systèmes temps réel doivent fournir des réponses qui sont, non seulement logiquement correctes, mais également temporellement correctes. Dans un système temps réel, il est très important que les tâches respectent leurs limites de temps. Dans de tels systèmes, un des problèmes de conception le plus délicat est la variabilité des temps d'exécution des tâches. En raison de cette variabilité, la plupart des algorithmes d'ordonnancement utilisés dans les systèmes temps réel monoprocesseur ou multiprocesseur sont basés sur le temps

d'exécution du pire cas (WCET – « Worst Case Execution Time ») des tâches. Le WCET des tâches est constant, par conséquent, les modèles de système en temps réel deviennent déterministes donc plus facile à comprendre et à mettre en application. Le problème de la conception basée sur le WCET est que, dans des applications temps réel avec une variabilité significative du temps d'exécution, l'analyse d'ordonnancement basée sur le temps d'exécution du pire cas mène à une faible utilisation des processeurs.

Dans ce projet, nous proposons quelques méthodes pour l'ordonnancement flexible des tâches du décodage Turbo qui sont très avantageuses en comparaison avec la méthode d'ordonnancement WCET. Les méthodes proposées sont inspirées des méthodes d'ordonnancement qui traitent des calculs flexibles. L'expression de calcul flexible se rapporte à une classe importante d'applications conçues et implémentés pour faire un compromis entre la qualité des résultats (services) qu'elles produisent et le temps et les ressources qu'elles emploient pour produire ces résultats. Plus spécifiquement, une application flexible peut réduire ses exigences de temps et de ressources aux dépens de la qualité de ses résultats tant que l'utilisateur trouve la qualité des résultats acceptable. Une application flexible peut dégrader graduellement sa qualité quand les ressources sont limitées et que les demandes en calculs sont élevées. Les méthodes d'ordonnancement proposées dans ce projet sont adaptées à la variabilité du processus de décodage Turbo et elles ajustent dynamiquement l'effort de traitement pendant l'exécution tout en gardant une qualité acceptable des résultats.

0.2 Flot de traitement dans une station de base UMTS

Dans cette section, nous décrivons brièvement les flots de traitement dans une station de base WCDMA compris dans la liaison descendante (« downlink ») et ascendante (« uplink »), qui correspondent respectivement aux côtés émetteur et récepteur de la station de base [29]. Dans ce projet, le flux de traitement de la liaison montante qui correspond au côté récepteur de la station de base est considéré.

0.2.1. Liaison descendante

Du côté de la liaison descendante, la station de base UMTS transmet un ensemble de canaux physiques communs aux utilisateurs dans la zone de portée de la station de base. La figure 2 (page 13) montre le flux de traitement de la station de base pour un utilisateur de la liaison descendante. Les données d'entrée venant de la couche du MAC (« Media Access Control ») se composent de différents flots (« streams ») de données. En premier lieu, le calcul de redondance cyclique (CRC – « Cyclic Redundancy Check ») et le codage de correction d'erreurs vers l'avant est ajouté aux flots. Ceux-ci sont alors envoyés par un ajusteur de taux (« rate matcher ») qui s'assure que le débit des trains est adapté à la couche physique. Les trains sont intercalés, segmentés dans des fenêtres et puis intercalées une autre fois. À la sortie, ils sont mappés, étalés au taux de chip (le chip est l'unité fondamentale de la transmission CDMA) et, finalement, envoyés à l'émetteur radio.

0.2.2. Liaison montante

Le flux de traitement de la liaison montante est semblable au flux de la liaison descendante mais il inclut beaucoup plus de calculs. La figure 3 (page 14) montre le flux de données pour le traitement de la liaison montante. Une combinaison par trajets multiples est d'abord effectuée, basée sur un filtre de recherche de trajets multiples et un récepteur Rake. Le récepteur Rake additionne les trajets multiples et effectue l'étalement (« despreading ») du signal d'entrée. Ensuite, la trame est désentrelacée et on effectue l'ajustement inverse des taux (« reverse rate matching »). Ensuite, la reconstitution des trames radio (« radio frame reassembly ») est faite, suivi d'un désentrelaceur différent. Par la suite, un décodage pour correction d'erreurs est employé pour reconstituer les données reçues qui sont envoyées à la couche de MAC.

0.3. L'assignation d'application sur une plate-forme multiprocesseur

Comme mentionné auparavant, nous avons l'intention d'implémenter le traitement des signaux d'un récepteur de station de base UMTS sur une plate-forme multiprocesseur. Dans un récepteur de station de base UMTS, les trames radio arrivent à un taux définissant une période de traitement, où chaque trame radio reçue est une concaténation de blocs de transport, où chacun est associé à un utilisateur donné. L'application cible se compose d'un traitement de liaison montante montré dans la figure 3 (page 14) qui doit être appliqué aux blocs de données reçues correspondant aux utilisateurs de la station de base.

La figure 27 (page 67) montre un exemple d'application UMTS servant 10 utilisateurs. Afin de simplifier l'exemple, la liaison montante traitant chaque utilisateur est décrite comme comportant trois segments de traitement. Pour implémenter l'application sur une plate-forme multiprocesseur, les segments de traitement doivent être mappés sur les processeurs. Comme premier exemple de stratégie d'assignation, nous supposons que les segments de traitement de chaque utilisateur sont assignés à différents processeurs. Ainsi, les segments en chaîne de chaque utilisateur peuvent être exécutés en mode pipeline sur différents processeurs. Aussi, nous considérons que chaque processeur exécute seulement un type de traitement (à savoir un segment) pour les multiples utilisateurs. La stratégie d'assignation proposée est similaire à la méthode présentée dans [29]. Un exemple d'une telle stratégie d'assignation est illustré à la figure 28 (page 67), qui correspond au cas où l'application de la figure 27 (page 67) est implémentée sur quatre processeurs.

0.4. Modélisation de la performance

Comme mentionné précédemment, nous devons concevoir un modèle de performance pour estimer le temps d'exécution des applications sur la plate-forme avant leur exécution. Dans ce but, nous proposons une méthodologie de modélisation de la

performance basée sur un modèle Matlab qui inclut les caractéristiques temporelles d'exécution de l'application et les primitives de la plate-forme à un niveau élevé. On suppose que les caractéristiques temporelles correspondant à l'exécution des différentes parties de l'application ont été extraites précédemment (à l'aide, par exemple, d'un simulateur de la plate-forme cible) et sont incluses dans le modèle de performance. Un tel modèle Matlab émule le temps d'exécution des différentes parties de l'application sans aucun test de fonctionnalité. De cette façon, le modèle fournit une évaluation rapide du temps d'exécution permettant également de valider les différentes stratégies pour l'assignation et l'ordonnancement de l'application sur la plate-forme. Le modèle permet également d'estimer la capacité de la plate-forme à soutenir les trafics demandés et il fournit une analyse statistique des services fournis par le système.

0.4.1. Étapes pour créer un modèle de performance

La méthodologie proposée pour la modélisation de performance inclut trois étapes. À la première étape, nous modélisons l'application qu'on désire mapper sur la plate-forme cible. Nous savons que l'application doit être divisée en plusieurs segments de traitement et que ces segments doivent être assignés aux différents processeurs de la plate-forme. À cette première étape, tous les segments de traitement sont présentés en tant que différentes tâches. Chacune de ces tâches inclut les paramètres temporels représentant l'exécution sur le processeur cible du segment de traitement correspondant. Également, les transmissions de données qui devront être affectées entre les processeurs sont modélisées par différentes tâches où sont inclus les paramètres temporels qui représentent les transmissions de données correspondantes. De cette façon, selon la stratégie d'assignation employée pour assigner les segments de traitement aux processeurs, nous créons un modèle de l'application mappée comprenant plusieurs tâches et l'information de dépendances entre ces tâches. La figure 9 (page 39) montre la première étape de notre stratégie de modélisation appliquée à un exemple simple.

À la deuxième étape, nous modélisons l'architecture de la plate-forme cible comme un ensemble d'unités afin de représenter les différents processeurs ainsi que la partie communication de la plate-forme. Après, selon la stratégie d'assignation, les tâches créées à la première étape sont assignées aux unités créées lors de la première phase de la deuxième étape. Il est important de mentionner que les tâches qui représentent les transmissions de données entre les processeurs sont assignées à l'unité qui représente la partie communication de la plate-forme. À la fin de la deuxième étape, le modèle créé inclut plusieurs unités représentant l'architecture de la plate-forme où chaque unité inclut à son tour plusieurs tâches représentant les paramètres temporels de traitement ou des transmissions de données. La figure 11 (page 41) montre le modèle créé à la deuxième étape de la modélisation pour le même exemple que celui dans la figure 9 (page 39). Lors de la simulation du modèle, les tâches sur les différentes unités doivent être exécutées selon un ordre qui est imposé par les relations de précédence entre les segments de traitement modélisés. Une tâche est prête à être exécutée lorsque l'exécution des tâches précédentes, sur la chaîne de la tâche, est terminée.

À la troisième étape de la modélisation, nous considérons une autre unité dans le modèle qui est responsable de contrôler l'exécution des tâches sur différentes unités. Cette unité supplémentaire s'appelle Maître (« Master ») et elle modélise un autre processeur de la plate-forme. Nous supposons que le Maître inclut un module appelé Synchroniseur (« Synchronizer ») pour définir les temps où les différentes tâches sont prêtes à être exécutées en vérifiant l'état d'exécution de toutes les tâches. Le Maître inclut également un module qui définit le temps de simulation. Le temps de simulation est utilisé pour calculer les paramètres temporels des tâches pendant la simulation. Les unités qui incluent les tâches s'appellent esclaves (« Slaves ») et elles sont responsables d'exécuter leurs tâches respectives. Nous supposons que chaque esclave inclut un module appelé Ordonnanceur (« Scheduler ») en plus de ses tâches. L'Ordonnanceur est responsable de choisir une tâche entre les tâches de l'esclave qui sont prêtes pour l'exécution basée sur sa stratégie d'ordonnancement. La tâche choisie par l'Ordonnanceur est exécutée par l'esclave. La figure 13 (page 43) démontre le modèle

de performance créé à l'étape finale de la modélisation pour le même exemple montré dans la figure 9 à la page 39.

De cette façon, nous créons un modèle de performance qui représente une description à niveau élevé de l'application ciblée et de l'architecture de la plate-forme. Pendant la simulation d'un tel modèle, toutes les unités mentionnées (Maître et esclaves) sont simulées. Nous considérons que la simulation est faite en plusieurs étapes. À chaque étape de la simulation, l'unité Maître est premièrement simulée alors que les Esclaves sont ensuite simulés les uns après les autres. La simulation de l'unité Maître produit les temps où les tâches sont prêtes à être exécutées. Aussi la Maître définit le temps de simulation à l'étape courante de la simulation. La simulation des unités esclaves peut mener à l'exécution de quelques tâches prêtes ainsi qu'une mise à jour de plusieurs des paramètres temporels concernant des tâches. La simulation d'un modèle de performance peut produire l'information temporelle de l'exécution des tâches et également de l'application entière.

0.5. Décodeur Turbo

Cette section se concentre sur un segment de traitement intensif de WCDMA appelé le décodeur Turbo qui est caractérisé par une variabilité substantielle du temps de traitement. Afin d'améliorer la robustesse de la transmission et de réduire au minimum le taux d'erreurs de bloc (BLER – « BLock Error Rate ») de l'application, la troisième génération de UMTS emploie la méthode de décodage Turbo. Les spécifications du procédé de décodage Turbo considéré dans ce projet sont comme suit. Le codeur convolutionnel de l'émetteur est basé sur un code convolutionnel parallèle concaténé (« Parallel Concatenated Convolutional Code – PCCC »), et inclut deux encodeurs convolutionnels récursifs systématiques identiques de 8 bits ainsi qu'un entrelaceur à taux de codage de 1/3 [28, 30]. Le décodeur Turbo a une structure itérative basée sur l'algorithme de « Maximum-A-Posteriori (MAP) » [30]. Le décodeur utilisé du côté de la réception inclut deux modules consécutifs de MAP qui constituent une des

itérations exigées par le décodage. Puisqu'une itération simple implique beaucoup d'étapes de traitement, un critère automatique d'arrêt de décodage (ASDC – « Automatic Stop Decoding Criterion ») est employé pour terminer le processus de décodage dès qu'un niveau de fiabilité acceptable du décodage est atteint (selon un paramètre spécifique). Le nombre d'itérations est aussi limité entre 2 et 8. Dans la section suivante, la variabilité de traitement d'un processus de décodage Turbo est caractérisée et discutée. Puis, utilisant le modèle de performance, quelques algorithmes d'ordonnancement, à être utilisés par les processeurs qui sont consacrés à ce genre de traitement sont validés.

0.5.1. Variabilité de traitement dans le décodeur Turbo

La variabilité du temps de traitement du décodage Turbo vient du nombre variable d'itérations de décodage exigées pour atteindre un niveau de correction d'erreur acceptable. Pour étudier la variabilité de traitement du décodage Turbo, nous avons utilisé un modèle complet de Simulink qui a été développé par notre équipe pour représenter exactement le processus entier de codage/décodage Turbo. Dans ce modèle, le canal de communication a été représenté par un bloc AWGN (« Additive White Gaussian Noise »), alors que le codeur et le décodeur ont été implémentés par plusieurs fonctions Matlab selon les caractéristiques précédemment mentionnées.

En simulant ce modèle Simulink, nous avons extrait le nombre d'itérations efficaces de décodage dans un décodeur Turbo qu'il faut exécuter selon les signaux reçus sous différents états du canal de communication. Étant donné la nature de l'algorithme et la manière dont il a été codé, le temps requis pour effectuer chaque itération de décodage par le processeur cible est constant pour une taille de trame donnée. Des résultats typiques pour le nombre d'itérations obtenues sous différents états du canal sont présentés dans la figure 29 (page 69). Les distributions de probabilité observées ressemblent à des distributions de Poisson. Notez que, étant donné que le nombre d'itérations est limité entre 2 et 8, la forme de la distribution est biaisée. Le

nombre moyen d'itérations a été également estimé selon différents états du canal tel que rapporté à la figure 30 (page 70). Cette figure montre une diminution du nombre moyen d'itérations de décodage lorsque l'état de canal s'améliore, autrement dit que Eb/N0 augmente.

0.5.2. Ordonnancement du processus de décodage Turbo

Comme précédemment expliqué, dans ce projet, nous étudions l'implémentation d'un récepteur d'une station de base UMTS sur une plate-forme multiprocesseur. Le flot de traitement dans une transmission en liaison montante, qui correspond à notre récepteur de station de base UMTS, est présenté à la figure 3 (page 14). Basé sur la stratégie d'assignation expliquée à la section 0.3, nous avons supposé que chaque processeur de la plate-forme est consacré à l'exécution d'un seul type de segment de traitement tel que le « Rake », le « Rate matching » ou le décodage Turbo. Le traitement de ces segments s'effectue sur les blocs de données reçues de différents utilisateurs. Dans cette section, nous nous concentrons sur les processeurs consacrés à effectuer le processus de décodage Turbo des blocs reçus et nous discutons des concepts d'ordonnancement sur ces processeurs. Afin de simplifier le problème, nous considérons l'ordonnancement du décodage Turbo sur seulement un processeur.

Nous supposons que toutes les trames reçues par le récepteur de la station de base UMTS considéré ont le même taux d'arrivée définissant ainsi une période de traitement unique. Pour déterminer le nombre de blocs de données (nombre d'utilisateurs) qui peuvent être assignés à un processeur au cours de chaque période de traitement, nous devons estimer le temps d'exécution des processus correspondants de décodage Turbo sur le processeur cible. Nous supposons que le temps d'exécution total des processus de décodage Turbo assignés ne doit pas dépasser la période de traitement.

Vu la variabilité de traitement significative du décodage Turbo, nous supposons que l'estimation basée sur le pire cas du temps d'exécution serait fortement inefficace. Afin d'améliorer l'utilisation des ressources, nous considérons que le temps assigné à

chaque processus de décodage Turbo est plus petit que celui du temps d'exécution du pire cas (WCET) et que nous pouvons par conséquent assigner un nombre plus élevé de blocs de données (nombre d'utilisateurs) à un processeur. Cependant, il est possible que, dans certains cas, les processus exigent plus que leurs fenêtres temporelles assignées nominalement, dues à une dégradation du canal qui peut mener à des problèmes d'ordonnancement. Pour résoudre ces problèmes, nous proposons quelques méthodes flexibles d'ordonnancement qui sont associées aux stratégies de contrôle qui peuvent limiter le traitement efficace afin de rencontrer les contraintes temporelles. La description des méthodes proposées est présentée dans les sections suivantes.

Pour étudier les méthodes d'ordonnancement, nous décrivons un modèle de performance comprenant une unité Maître et un esclave basés sur les concepts expliqués dans la section 0.4. Le Maître, comprenant le module de synchronisation, correspond à un processeur et l'esclave correspond à un autre processeur de la plate-forme. L'esclave est responsable des tâches représentant les processus de décodage Turbo sur les blocs de données de différents utilisateurs. La modélisation de performance du décodage Turbo n'est pas basée sur le décodage détaillé, mais plutôt sur les distributions de service précédemment rapportées (figure 29 à la page 69) qui ont été obtenues par le décodage détaillé. Chaque tâche représentant le décodage Turbo contient un générateur de nombres aléatoires qui modélise les distributions mentionnées. La simulation de chaque tâche de décodage Turbo fournit le nombre efficace d'itérations de décodage (*it_eff*) et, par conséquent, le temps d'exécution du processus correspondant au décodage sur le processeur cible.

Mentionnons que, lorsqu'un bloc de données est reçu par le processeur pour être décodé, la tâche modélisant le processus est censée être activée. En d'autres termes, la tâche devient prête. Aussi, toutes les tâches assignées à l'esclave ont la même période d'activation qui est égale à la période de traitement. Nous supposons que toutes les tâches deviennent actives ou prêtes en même temps pour chacune des périodes de traitement.

0.5.2.1. Ordonnancement « one shot »

Premièrement, nous supposons qu'un certain nombre de tâches de décodage Turbo sont assignées à l'esclave. Ainsi, basées sur le nombre de tâches assignées à l'esclave et sur la période de traitement, les ressources consacrées pour exécuter le décodage par chaque tâche sont déterminées. Étant donné l'allocation des ressources pour chaque tâche de décodage Turbo, un budget d'itérations de décodage est garanti pour le décodage de chaque bloc. Ceci est déterminé par le système sous le nom de « Iteration Budget (*IB*) ». Au cours d'une période de traitement, l'Ordonnanceur placé sur l'esclave choisit une tâche à être simulée parmi toutes les tâches prêtes de l'esclave. Cette tâche est choisie à tour de rôle (« round robin »). Après la simulation de chaque tâche de décodage Turbo, si le nombre efficace d'itérations de décodage (*it_eff*) est plus grand que son *IB*, le nombre d'itérations émulés de décodage pour le bloc correspondant est considéré égal à *IB*. Le bloc auquel l'Ordonnanceur n'a pu assigner son *it_eff* avant la fin d'une période d'activation doit être considéré comme étant partiellement décodé.

Si le budget assigné est égal ou plus grand que *it_eff*, le nombre d'itérations émulées est considéré égal au *it_eff* correspondant. Un bloc qui peut atteindre son *it_eff* est entièrement décodé. En effet, si *it_eff* est inférieur au budget pour une tâche, la différence est distribuée entre les tâches non simulées dans la période de traitement courante et est ajoutée à leurs budgets assignés précédemment. Cette redistribution est faite aussi uniformément que possible. La figure 31 (page 75) démontre la méthode d'ordonnancement « one shot » sur un esclave comprenant quatre tâches de décodage Turbo au cours d'une période de traitement. Suivant les indications de cette figure, des blocs de données correspondants aux tâches 3 et 4 sont complètement traités, tandis que le décodage correspondant aux deux autres tâches est dégradé afin de respecter les budgets de ressource assignés.

0.5.2.2. Ordonnancement progressif (« gradual »)

Comme expliqué pour la méthode d'ordonnancement « one shot », au cours de chaque période de traitement, la partie non utilisée du budget de chaque tâche simulée de décodage Turbo est distribuée et ajoutée aux budgets assignés des tâches non simulées. De cette façon, les *IB* (« Iteration Budget ») des tâches qui sont simulées plus tard ont une plus grande possibilité d'augmentation comparativement aux tâches simulées plus tôt. En raison de la qualité aléatoire du décodage des blocs, il est possible que les tâches simulées postérieurement n'emploient pas totalement leur *IB* accru tandis que les tâches simulées précédemment auraient eu besoin de plus grands budgets.

Afin d'optimiser la répartition des ressources aux tâches, nous proposons une autre méthode appelée ordonnancement progressif. Dans cette méthode, un *IB* global est assigné à toutes les tâches qui est égal au nombre d'itérations de décodage qui pourraient être effectuées par le processeur cible au cours d'une période de traitement. Dans la méthode d'ordonnancement progressif, l'Ordonnanceur choisit à tour de rôle les tâches prêtes pour la simulation. Après l'émulation d'une itération de décodage pour chaque tâche choisie, la simulation de la tâche est suspendue (« preempted ») et, par conséquent, la prochaine tâche prête est choisie pour être simulée de la même manière.

La figure 33 (page 77) présente la méthode d'ordonnancement progressive sur un esclave comprenant quatre tâches de décodage Turbo au cours d'une période de traitement. Suivant les indications de la figure 33 (page 77), la valeur initiale de l'*IB* global dans l'exemple montré est égale à 17, ce qui est équivalent à la somme des *IB* initiaux des tâches dans l'exemple d'ordonnancement « one shot ». Après l'émulation d'une itération de décodage de chaque tâche, l'*IB* global est décrémenté par un. La simulation des tâches qui atteignent leur nombre efficace d'itérations de décodage est terminée. La simulation des tâches prêtes continue de la même manière jusqu'à ce que le budget global devienne zéro. Après l'arrêt dû à l'*IB* global, la simulation des tâches prêtes qui n'ont pas atteint à leur traitement efficace est arrêtée. Ceci mène à la

dégradation du traitement des blocs correspondants. Dans l'exemple montré dans la figure 33 (page 77), comme pour l'exemple « one shot », les décodages effectués pour la tâche 1 et la tâche 2 sont dégradées. Mais, dans cet exemple, une itération de plus est émulée pour les tâches mentionnées comparativement à l'exemple montré dans la figure 31 (page 75).

Dans la méthode d'ordonnancement progressif, le nombre d'itérations émulées pour chacune des tâches prêtes à chaque moment de la simulation est approximativement identique. Ainsi, les blocs correspondants aux tâches arrêtées, donc qui n'ont pas atteint leur nombre efficace d'itérations, ont presque le même niveau de traitement et par conséquent leurs traitements sont dégradés de manière presque identique.

0.5.2.3. Ordonnancement « one shot » par priorité

Comme expliqué dans l'ordonnancement « one shot », les tâches simulées plus tôt ont la possibilité de recevoir de plus petits *IB*, ce qui cause des erreurs résiduelles plus prononcées en raison de la dégradation de traitement émulé, en comparaison aux tâches simulées postérieurement. Puisque les tâches prêtes sont choisies et simulées dans un ordre fixe pour toutes les périodes de traitement, les erreurs résiduelles associées aux tâches simulées plus tôt sont plus importantes que pour les tâches qui sont simulées plus tard.

De cette façon, les différences entre les qualités de service fournies pour les blocs (utilisateurs) correspondants augmentent en fonction du temps. Afin d'empêcher l'augmentation de la différence entre les qualités du service de l'utilisateur, nous faisons une modification dans la méthode d'ordonnancement « one shot ». Dans la méthode modifiée qui s'appelle ordonnancement « one shot » par priorité, nous assignons des priorités aux tâches prêtes pour déterminer leur ordre de simulation au cours de chaque période de traitement. Dans le modèle, nous avons considéré un paramètre appelé *sum_add_err,* qui est assigné à chaque tâche et qui représente les bits accumulés en

erreur additionnelle insérée dans le bloc correspondant, du commencement de la simulation jusqu'à la période de traitement courant.

Dans la méthode d'ordonnancement « one shot » par priorité, les budgets (*IB*s) sont assignés aux tâches de la même façon que dans la méthode « one shot ». La méthode pour la mise à jour du *IB* pour chaque tâche est également la même que pour l'ordonnancement « one shot » comme démontré dans la figure 31 (page 75). Avec cette méthode, l'ordre pour placer les tâches sur les colonnes montrées dans la figure 31 (page 75) est basé sur la priorité des tâches. À chaque période de traitement, les tâches prêtes ayant une valeur plus élevée de *sum_add_err* se voient assigner une priorité plus basse pour la simulation et sont placées sur les colonnes qui sont plus près de la dernière colonne. De cette façon, les tâches qui sont caractérisées par des taux d'erreurs accumulées plus élevés ont la possibilité de recevoir plus de ressource dans la période de traitement en cours. Ainsi, les qualités de service fournies pour différents utilisateurs deviennent plus uniformes comparativement à la méthode d'ordonnancement « one shot ».

0.5.2.4. Ordonnancement progressif par priorité

Dans l'ordonnancement progressif, les tâches simulées postérieurement dans différents niveaux d'itération ont plus de possibilité de ne pas recevoir de ressource lorsque le *IB* global est terminé. Ainsi, plus d'erreurs additionnelles sont associées aux blocs décodés plus tard, en raison de la dégradation de traitement émulée, comparativement aux blocs décodés plus tôt. Puisque les tâches prêtes sont simulées dans un ordre fixe pour tous les niveaux d'itération et toutes les périodes de traitement, les erreurs additionnelles associées aux tâches simulées postérieurement à différents niveaux d'itération augmentent avec le temps beaucoup plus rapidement que les tâches qui sont simulées plus tôt. De cette façon, les différences entre les qualités de service fournies aux utilisateurs sont augmentées avec le temps.

Afin d'empêcher l'augmentation des différences entre les qualités de service pour chaque utilisateur, nous faisons une modification à la méthode d'ordonnancement progressif. Dans la nouvelle méthode d'ordonnancement modifiée, qui s'appelle ordonnancement progressif par priorité, comme pour l'ordonnancement progressif, les tâches sont simulées graduellement et un *IB* global est assigné pour limiter le traitement émulé des tâches comme démontré dans la figure 33 (page 77). En plus, comme pour la méthode « one shot » par priorité, le paramètre *sum_add_err*, est estimé et associé à chaque tâche totalement simulée dans chaque période de traitement qui définit la priorité de la tâche dans la période suivante.

Autrement dit, dans cette méthode, l'ordre de placement des tâches sur les colonnes montrées dans la figure 33 (page 77) est basé sur la priorité des tâches. À chaque période de traitement, les tâches prêtes avec une valeur plus élevée de *sum_add_err* ont une priorité plus élevée pour la simulation et sont placées sur les colonnes qui sont plus près de la première colonne. De cette façon, les tâches qui sont caractérisées par des taux des bits en erreur accumulés plus élevés ont la possibilité de recevoir plus de ressources dans la période de traitement courante. Ainsi, les qualités de service fournies aux différents utilisateurs deviennent plus uniformes comparativement à la méthode d'ordonnancement progressif.

0.5.3. Résultats de simulation

Nous avons implémenté le modèle de performance expliqué dans la section 0.4 comprenant un Maître et une unité esclave. Les méthodes d'ordonnancement proposées dans la section précédente sont également incluses dans l'unité esclave en tant que différentes options pour ordonnancer les tâches de décodage Turbo. Le modèle a été simulé dans différents cas de figure afin de tester les méthodes d'ordonnancement proposées sous différents états de canal et charges pour des durées appropriées. La simulation du modèle a fourni le taux des bits en erreurs (BER) associés aux blocs décodés dans chacune des périodes de traitement. Afin de simplifier les résultats, nous

avons calculé le BER moyen de toutes les tâches correspondantes pendant la simulation selon différents cas de charges du système et d'états de canal. Les tâches spécifiques de décodage Turbo correspondent aux blocs de 656 bits de données (avant le codage donc (3*656) +12 =1980 symboles après codage), et ces trames sont transmises à 64 k symboles/s pour des périodes de 40 ms.

Étant donné la période des trames reçues et le pire temps de traitement du décodage Turbo sur le processeur cible qui comprend 8 itérations de décodage dans le cas de la conception WCET, les processus de décodage Turbo pour 14 utilisateurs peuvent être assignés au processeur. Autrement dit, dans notre modèle de performance, selon la méthode du WCET, nous pouvons assigner 14 tâches de décodage Turbo à l'esclave sans causer de dégradation de traitement. Le cas consistant à assigner 14 utilisateurs (méthode du WCET) est employé comme une référence pour toute comparaison avec les méthodes d'ordonnancement proposées où plus d'utilisateurs pourront être assignés.

0.5.3.1. Ordonnancement « one shot »

En simulant le modèle de performance mentionné suivant la méthode d'ordonnancement « one shot », le BER moyen de toutes les tâches (correspondant aux différents utilisateurs) est obtenu selon le nombre d'utilisateurs assignés et l'état du canal et est montré à la figure 35 (page 82). La distance horizontale entre les courbes du BER moyen et de la courbe de référence (cas de 14 utilisateurs) pour un BER précis, démontre la dégradation moyenne du gain de décodage. Basé sur la figure 35 (page 82), la dégradation moyenne du gain de décodage est obtenue pour 3 cas où diffèrent le nombre d'utilisateurs pour une valeur de BER de $2*10^{-5}$. Ces résultats sont rapportés dans le tableau 4 qui démontre qu'avec 29 utilisateurs assignés à un processeur, la dégradation de gain de décodage est d'approximativement 0.15 dB. Une telle valeur de dégradation est négligeable et n'exerce aucun effet significatif sur la qualité du service.

0.5.3.2. Ordonnancement progressif

En simulant le modèle de performance mentionné suivant l'algorithme d'ordonnancement progressif, le BER moyen des utilisateurs est obtenu pour un nombre variable d'utilisateurs assignés et pour différents états de canal et est montré à la figure 36 (page 84). Basé sur les figures 35 et 36 (pages 82 et 84), l'amélioration des BER moyen en utilisant la méthode d'ordonnancement progressif comparativement à la méthode d'ordonnancement « one shot » est remarquable. En examinant la figure 36 (page 84), la dégradation moyenne du gain de décodage dans 5 cas de figure avec un nombre d'utilisateurs différent et une valeur de BER de $2*10^{-5}$ est obtenue et présentée dans le tableau 5. Ce tableau montre que la dégradation moyenne du gain de décodage pour les cas de 23 et 29 utilisateurs est zéro tandis que nous avions une dégradation de gain pour le même nombre d'utilisateurs avec la méthode d'ordonnancement « one shot ». En se basant également sur le tableau 5, la dégradation moyenne du gain de décodeur pour plus d'utilisateurs, tels que 32 et 35, est négligeable.

0.5.3.3. Ordonnancement « one shot » par priorité et progressif par priorité

En utilisant l'ordonnancement « one shot » par priorité et puis l'ordonnancement progressif par priorité dans notre modèle de performance et en simulant ces modèles correspondants, le BER moyen des utilisateurs selon le nombre d'utilisateurs assignés et l'état du canal sont obtenus. En observant les résultats, nous nous sommes rendu compte que les résultats moyens des BER obtenus en employant la méthode « one shot » par priorité sont identiques aux résultats correspondants dans le cas de méthode « one shot » montrée à la figure 35 (page 82). La méthode progressive par priorité fournit également les mêmes résultats de BER moyens que ceux produits par la méthode progressive et sont affichés à la figure 36 (page 84).

Afin de démontrer l'avantage des méthodes par priorité à uniformiser les qualités de service des utilisateurs, nous avons également estimé la variance du BER des utilisateurs. La variance du BER des utilisateurs selon les différentes méthodes d'ordonnancement proposées sont présentés dans les figures 37, 38, 39 et 40 (pages 85,86 et 87). Ces courbes sont paramétrées en fonction du nombre d'utilisateurs. Dans les figures 39 et 40 (pages 86 et 87), les variances du BER pour les valeurs Eb/N0 qui sont supérieures aux valeurs montrées par les lignes pointillées sont égales à zéro. En observant les figures 37 et 38 (pages 85 et 86), nous nous rendons compte que la méthode « one shot » par priorité uniformise les performances des BER des utilisateurs en obtenant des variances plus petites comparativement à la méthode « one shot ». En comparant également les figures 39 et 40 (page 86 et 87), on peut prouver que la méthode d'ordonnancement progressif par priorité rend plus uniforme les performances des BER des utilisateurs en fournissant des variances plus petites comparativement à la méthode progressive.

Étant donné les résultats affichés pour les BER moyen et la variance des BER, nous trouvons que l'ordonnancement progressif par priorité est la méthode la plus efficace entre les méthodes proposées puisqu'elle fournit la meilleure performance pour le BER et offre une uniformité de service pour plusieurs utilisateurs.

0.6. Conclusion

Dans ce projet, un modèle dynamique a été présenté pour évaluer l'exécution d'une application WCDMA sur une plateforme MPSoC. Plus spécifiquement, nous nous sommes concentrés sur le décodeur Turbo, une partie de l'application demandant un grand effort de calcul et présentant une variabilité de traitement substantielle. Notre modèle nous a permis de dériver et valider quelques méthodes flexibles pour l'ordonnancement des tâches de décodage Turbo, qui sont adaptées à l'effort de traitement variable exigé pour le décodeur. En employant le modèle de performance

présenté, l'efficacité de ces méthodes d'ordonnancement a été démontrée, ce qui a également justifié l'utilisation de notre modèle d'évaluation de performance.

Toutes les méthodes d'ordonnancement flexibles proposées, une fois comparées à la méthode d'ordonnancement du temps d'exécution du pire cas (WCET), améliorent l'utilisation des processeurs en employant une évaluation plus juste de l'effort de décodage et en causant une dégradation de traitement acceptable. La différence entre les méthodes d'ordonnancement flexible proposées était dans l'uniformité de la qualité de service fournit pour les utilisateurs. La dernière méthode d'ordonnancement flexible appelée ordonnancement progressif par priorité a fourni la qualité de service la plus uniforme pour les utilisateurs. Cette méthode, une fois comparée à une méthode d'ordonnancement du temps d'exécution du pire cas (WCET), a permis d'augmenter le nombre d'utilisateurs de 14 à 35, alors que la conservation d'une qualité du service acceptable se reflétait dans une dégradation très petite de moins de 0.1 dB de gain de décodage.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SIGNS ANDABBREVIATIONS

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
ave_add_err	Average number of additional bits in error
BER	Bit Error Rate
CDMA	Code-Division Multiple-Access
CMOS	Complementary Metal–Oxide–Semiconductor
Com	Communication part
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DDR	Double Data Rate
Dep	Vector that describes dependencies between tasks
DMA	Direct Memory Access
DPCCH	Dedicated Physical Control Channel
DPDCH	Dedicated Physical Data Channel
DS-CDMA	Direct-Sequence Code-Division Multiple-Access
DSP	Digital Signal Processor
DSSS	Direct-Sequence Spread Spectrum
EDF	Earliest-Deadline-First
F	Function
FBI	Feedback Information
FDD	Frequency Division Duplex
fin	Vector that describes the completion time of tasks
FPGA	Field-Programmable Gate Array
FS	Flexible Scheduling
GMII	Gigabit Media Independent Interface
GPIO	General Purpose Input/Output

HDLC	High-Level Data Link Control
HW	Hardware
IB	Iteration Budget
IDE	Integrated Development Environment
it_eff	Effective number of iterations
it_max_perm	Maximum permitted number of iterations
LLR	Log-Likelihood-Ratio
load_unit	Vector that describes loading time of processors
MAC	Medium Access Control
MAP	Maximum-A-Posteriori algorithm
MII	Media Independent Interface
MIPS	Million Instructions Per Second
MPSoC	Multi-Processor System-on-Chip
NoC	Network-on-Chip
OCT1010	Octasic's platform including 15 Opus cores
OPERA	Octasic Polytechnique ETS Radio Application
P	Processor
PBMT	Performance Based Modeling Tool
PCCC	Parallel Concatenated Convolutional Code
RACH	Random Access Channel
rdy	Vector that describes ready time of tasks
RM	Rate-Monotonic
RMII	Reduced Media Independent Interface
PRACH	Physical Random Access Channel
RRC	Radio Resource Control
RSC	Recursive Systematic Convolutional
RS232	Recommended Standard 232
RTL	Register Transfer Level
SCH	Synchronization Channel

SF	Spreading Factor
SoC	System-on-Chip
SPI	Serial Peripheral Interface Bus
SR	Synchronous Reactive
SR MoC	Synchronous Reactive Model of Computation
st	Vector that describes the earliest time for tasks' execution
start	Vector that describes the execution starting time of tasks
sum_add_err	Accumulated additional bits in error
SW	Software
T	Task
TDD	Time Division Duplex
TDM	Time-Division Multiplexing
t_exe	Execution time of a task
TFCI	Transport Format Combination Identifier
TLM	Transaction Level Modeling
TPC	Transmit Power Control
tsk_unit	Vector that describes assignment of tasks to processors
t_sim	Simulation time
u	Utilization of a task
U	Total utilization
UMTS	Universal Mobile Telecommunication System
Utopia	An ATM protocol evolved into System Packet Interface
V	Vector
VPC	Virtual Processing Components
VPU	Virtual Processing Unit
WCDMA	Wide-band Code Division Multiple Access
WCET	Worst Case Execution Time
3GPP	3rd Generation Partnership Project

CHAPTER 1

INTRODUCTION

Multi-Processor System-On-Chip (MPSoC) platforms can provide high computational performance along with load balancing on a network of processors. Such architectures are appropriate targets for the implementation of dynamic and computationally intensive applications. High-performance multi-core Digital Signal Processors (DSPs) are increasingly used for telecommunication equipments to process voice, video, and radio signals. Since DSPs are software-driven, when sufficiently powerful, they provide more flexibility than dedicated ASICs. Considering these explanations, a multi-DSP platform is chosen as the target architecture in this project.

The target application in this project is the WCDMA (Wide-band Code Division Multiple Access) process on the received data blocks corresponding to different users in a Universal Mobile Telecommunication Systems (UMTS) receiver base-station. The WCDMA makes possible that different users share a relatively wide spectral band using coding instead of time slots. The WCDMA modulator is based on the spread spectrum modulation technique which consists in multiplying the lower-rate data stream with a higher rate (known as the chip sequence). In this project, we intend to implement a UMTS receiver base-station including the corresponding WCDMA processes on a multi-DSP platform.

Considering the real-time and dynamic characteristics of the target application, and also the complexity of MPSoC design, it is necessary to have means of validating the capacity (e.g. the traffic it can serve) and performance of the platform before implementation, either to verify if a proposed MPSoC architecture is suitable for the application or to determine the number of processors required to fulfill the requirements. For these reasons, we have developed a dynamic modeling strategy for evaluating the performance of MPSoC designs based on high level models of the application and of the

architecture. At this level of modeling, the application is represented as a set of tasks that are to be performed, their resource requirements, such as execution times, release time, etc., and the order in which the tasks are to be executed. Also, the architecture upon which the application will execute is represented simply as a set of computational resources and a communications fabric for transferring data between them. One outstanding aspect of this model is that the functionality is retrieved directly from the model, which speeds up characterization and system design. Such performance modeling allows quick validation of the efficiency of strategies for mapping and scheduling a complex application on the target platform before run time. The developed modeling strategy can be utilized for any MPSoC design and it is not limited to the target application of this project.

Afterward, we propose a method for mapping the different parts of the WCDMA application on the platform. We focus specifically on the Turbo decoding process which is a computationally intensive part of the WCDMA application and scheduling of this kind of process is investigated in details. The Turbo decoding process is characterized by a significant variability of the processing effort which makes the scheduling of such process more critical. It is well known that real-time systems must provide responses which are not only logically correct, but also temporally correct. In a real-time system, it is very important that the tasks meet their deadlines. In such systems, one of the most delicate design problems is the variability of tasks execution time. Due to such variability, most scheduling algorithms used in uniprocessor or multiprocessor real-time systems are based on the Worst Case Execution Time (WCET) of application tasks. The WCET of tasks are constant, consequently the real-time system models become deterministic, thus easier to understand and implement. The problem of such WCET based design is that, in real-time applications with a significant variability of the execution time, the scheduling analysis based on WCETs leads to low processor utilization. In this project, some flexible scheduling methods are proposed for Turbo decoding tasks which are highly advantageous comparing to the WCET scheduling method. The proposed methods are inspired from the scheduling methods that deal with

flexible computations. The term flexible computation (or application) refers to a wide class of applications that are designed and implemented to trade off, at run-time, the quality of the results (services) they produce with the amount of time and resources they use to produce the results. In particular, a flexible application can reduce its time and resources demands at the expense of the quality of its results. For as long as the user finds its quality result acceptable, a flexible application can degrade gracefully when resources are limited and the demands of competing workloads are high. The proposed scheduling methods in this project are adapted to the variability of the Turbo decoding process and they dynamically adjust the processing effort during the execution while keeping an acceptable quality of results.

There are several references on performance modeling methods in the literature such as [10, 13, 15 and 25]. In this project, similarly to the mentioned references, a performance model is structured, which is used to verify the execution time of applications on the platform before implementation. The mentioned model has been created using the Matlab/Simulink software and it is based on the structure of the target application and platform. Unlike the proposed tool in [15], in this project, we consider only the performance modeling while the developed environment is not used for design purposes. Also, the modeling methods presented in [10, 25] are SystemC-based where, in this project, the presented model is developed using the Matlab/Simulink software which is the same environment that is used in our project to model and characterize the target application.

Also, research has been published on the mapping and scheduling of Turbo decoding on an MPSoC platform such as [6, 8, 18, 19 and 20]. In this project, similarly to [6] and [18], scheduling of the Turbo decoding process for several encoded blocks on the dedicated processors is studied. However, our scheduling methods allow much more flexible degradation by considering dynamic iteration budgets, when compared to the decoding degradation presented in [18]. Such processing degradation concept is not considered in [6]. Unlike [8], [19] and [20], we consider that the Turbo decoder

algorithm consists of only one monolithic task and the Turbo decoding process on each coded block is performed solely on one processor. Thus, the Turbo decoding process on the individual processors is data independent, which reduces the data communications between the processors. By allocating different encoded blocks on the individual processors, we also exploit the platform parallelism.

This project is part of a project called OPERA which is a collaboration between Octasic semiconductor, École Polytechnique de Montréal and École de Technologie Supérieure de Montréal. The objective of the OPERA project is to develop a methodology for automatic mapping of a Simulink system-level model of the target application to the Vocallo multi-DSP platform (platform designed and fabricated by Octasic). Firstly, the target application must be modeled in the Simulink environment, using C/C++, Matlab scripts or Simulink library blocks. Thus, using several tools, we intend to effectively convert such a high-level Simulink model into low-level code executable by the Vocallo DSP cores. Moreover, we will perform resource requirement estimations which will let us determine an efficient strategy for mapping the obtained low-level code of the application on the platform processors. Besides the mentioned conversion process, the developed performance modeling methodology from my project allows rapid execution performance verification to validate several mapping and scheduling methods without any functional verification.

The rest of this thesis is organized as follows. Some basic concepts and a literature review regarding our project are presented in Chapter 2. We first describe the UMTS base-station and the WCDMA processing. Then, we describe the architecture of the Vocallo multi-DSP platform which is used as the target hardware in this project. Afterward, we present a literature review on performance modeling methods. Also, a literature review is presented on mapping system-level models into MPSoC platforms which describe the methods to combine system-level models and architecture-level descriptions for MPSoC designs. Then, we present some basic concepts on multiprocessor scheduling and synchronization. Also, we explain the WCET-based

design and the advantages and disadvantages of this method in a real-time system implementation. Afterwards, we present some methods for scheduling flexible applications which allow trading the quality of results for the amount of processing time and the required resources. At the end of Chapter 2, we introduce a literature review on the mapping and scheduling of Turbo decoding in an MPSoC platform.

In Chapter 3, the basic concepts of our proposed performance modelling methodology and several steps of providing a performance model are described. The proposed performance modelling methodology includes three steps of modelling to create a final performance model of a given application and platform. In the first step, we model the application which must be mapped to the target platform. The created model at this step includes a chain of tasks representing different processing and also the data transmissions in the platform. In the second step, we model the architecture of the target platform as a set of units to represent different processors and also the communication part of the platform. At the end of the second step, the created model includes several units that represent the architecture of the platform where each unit includes several tasks that represent the timing parameters of the processing or data transmission. At the third modelling step, we consider another unit in the model which is responsible for managing or synchronizing the tasks execution on different units.

In Chapter 4, we first discuss the concept of mapping the uplink WCDMA processing corresponding to a UMTS base-station receiver on an MPSoC platform. Afterward, we focus on the processors of the platform which are dedicated to perform the Turbo decoding process. Then, the processing variability of the Turbo decoding is discussed and the BER performance of such decoding is characterized. Then, we discuss the processor scheduling methods dedicated to this process. We propose four flexible scheduling methods which are adapted to the variable characteristics of the Turbo decoding. In the proposed flexible methods, the resources assigned to each decoding process are variable and some processing degradations may be imposed to the processes in order to meet the timing constraints. To investigate the proposed methods, we utilize

our developed performance modelling methodology and we create a performance model including the developed flexible scheduling methods and the tasks representing the Turbo decoding processes. Simulating such a model provides the BER performance results for the decoded blocks for the different scheduling methods. Comparing the obtained results with a Worst Case Execution Time (WCET) design shows the advantage of the proposed flexible scheduling methods in terms of improved processor utilization. Finally, we present the elapsed simulation times in two cases of functional and performance modelling of Turbo decoding.

In Chapter 5, a conclusion is given on the presented thesis and we summarize the whole work and the obtained results. We present briefly the advantage of all proposed scheduling methods compared to a WCET scheduling method to improve the processors utilization, by indicating the significant increase in number of supported users and the negligible amount of decoding degradations. Then, we present the advantage of utilizing our developed performance model by indicating the ratio of simulation times in two cases of functional and performance modeling. Finally, we present the future work of this project.

CHAPTER 2

BASIC CONCEPTS AND LITERATURE REVIEW

2.1. Universal Mobile Telecommunication System (UMTS)

In this section, some basic concepts of the third generation mobile communication systems, called Universal Mobile Telecommunication System (UMTS), and the involved processing are introduced. These third generation systems are designed for multimedia communications to provide person-to-person communications with high quality images and video, while access to information and services on public and private networks can be enhanced by high data rates and new flexible communication capabilities. The access scheme for UMTS is Direct-Sequence Code-Division Multiple-Access (DS-CDMA). The information is spread over a band of approximately 5 MHz. This wide bandwidth has given rise to the name Wideband CDMA or WCDMA. This scheme supports two different modes namely:

1- Frequency Division Duplex (FDD): The uplink and downlink transmissions employ two separated frequency bands for this duplex method. A pair of frequency bands with specified separation is assigned for each connection.

2- Time Division Duplex (TDD): The Uplink and downlink transmissions are carried over the same frequency band using synchronized time intervals. Therefore, time slots in a physical channel are divided into transmission and reception parts.

In the conventional UMTS terminology, the downlink transmission refers to the transmission from a radio base station to one user and the uplink transmission refers to the transmission from a user to the base station.

WCDMA allows different users to share a relatively wide spectral band using coding instead of time slots. The WCDMA modulator is based on the spread-spectrum modulation technique which consists in multiplying the lower rate data stream with a higher rate one (known as the chip sequence). The baseband data spectrum is spread according to a processing gain which is called the spreading factor (SF). In this way, each user is allocated a spreading sequence used to transmit its narrowband data signal over the broader spectral band. Each user is differentiated from other users by the given spreading sequence which preferably should be orthogonal to the other spreading sequences in use. The narrow-band signal can be recovered at the receiver using the same mechanism.

WCDMA is the main third generation air interface in the world and is deployed in many countries. The specification of WCDMA has been created in the 3rd Generation Partnership Project (3GPP), which is a joint project of standardisation bodies from Europe, Japan, Korea, the USA, and China. In the rest of this section, we provide a description of layer 1 (also called the physical layer) of the radio access network of WCDMA systems operating in the FDD mode [11].

2.1.1. WCDMA Physical layer

The physical layer, Medium Access Control (MAC) layer, and Radio Resource Control (RRC) layer, which are called respectively layer 1, 2 and 3, are three principal layers of a radio interface [27]. The physical layer is a fundamental layer upon which all higher layers are based. It provides the physical signals to transmit the data. This layer assures the appropriate preparation of data for transmission by applying the coding and modulating operations. Since implementing the operations in the physical layer is the case study of this project, we briefly describe the channel structure in this layer. WCDMA defines several physical channels in both the downlink and the uplink placed on the physical layer:

· The Dedicated Physical Data Channel (DPDCH) is used to carry dedicated data generated at layer 2 and above.

· The Dedicated Physical Control Channel (DPCCH) carries layer 1 control information.

Each connection is allocated one DPCCH and zero, one, or several DPDCHs. In addition, there are some common physical channels defined as:

· Primary and secondary Common Control Physical Channels (CCPCH) to carry downlink common channels.

· Synchronization Channels (SCH) for cell search.

· Physical Random Access Channel (PRACH) to carry the RACH (Random Access Channel). The RACH is used in wireless access terminals such as mobile phones when it needs to get the attention of a base station in order to initially synchronize its transmission with the base station.

2.1.2. Frame structure for uplink DPDCH/DPCCH

Communications between a base station and users require a time reference in order to synchronize the physical connection. In a WCDMA air interface, time is divided into radio frames of 10 ms (38400 chips) which are numbered from 0 to 4095. It is mentioned that a chip is a pulse of a Direct-Sequence Spread Spectrum (DSSS) code, such as a pseudo-noise code sequence used in Direct-Sequence Code Division Multiple Access channel access techniques. In a binary direct-sequence system, each chip is typically a rectangular pulse of +1 or −1 amplitude, which is multiplied by a data sequence (similarly +1 or −1 representing the message bits) and by a carrier waveform to make the transmitted signal.

Each radio frame is then subdivided into 15 radio slots (2560 chips). The data stream, or DPDCH, is mapped to the radio slots. The data bit stream lengths vary from

10 bits, for a Spreading Factor (SF) of 256, up to 640 bits for a SF of 4. The control stream, or DPCCH, is also mapped to the radio slot. The associated bit stream length is always 10 since the SF is set at 256. Up to 8 bits are reserved for the pilot sequence to be used in the channel estimation algorithm. Additional control bits are transported in this frame for physical layer purposes. For example, the Transmit Power Control (TPC) bits are responsible for controlling the power of the signal to be transmitted. The Transport Format Combination Identifier (TFCI) informs the receiver of the current structure of the transmitted transport channel. Also, Feedback Information (FBI) bits are to be used to support techniques requiring feedback. Fig. 1 shows the frame structure uplink DPDCH/DPCCH. [31].

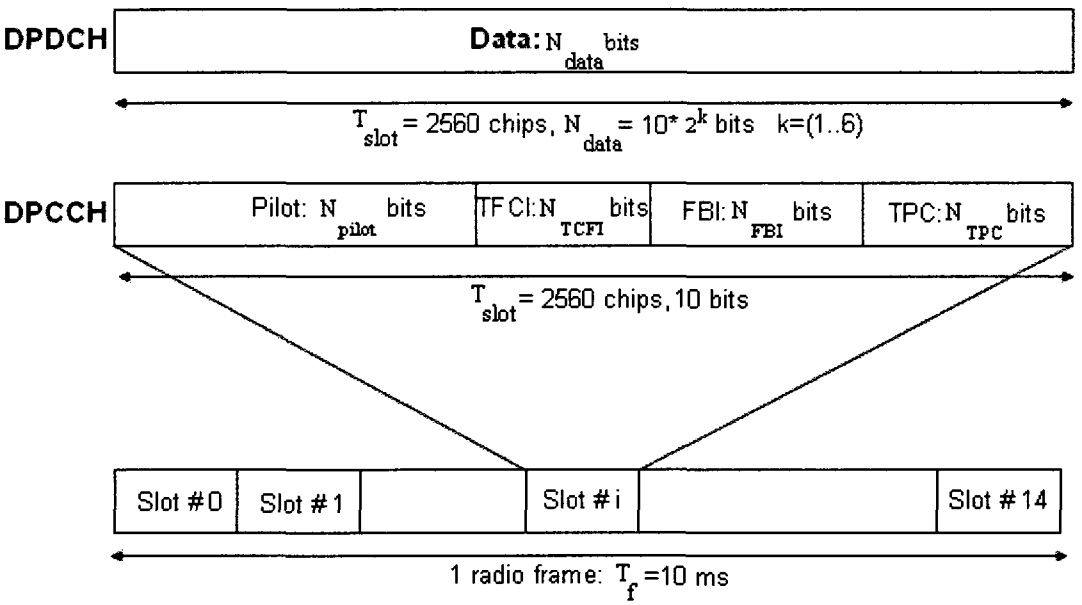

Figure 1 Radio frame structure for uplink DPDCH/DPCCH [31]

2.1.3. Processing in a WCDMA/FDD radio base station

A description is presented on the processing flows performed in both downlink and uplink parts of a WCDMA/FDD radio base station which is based on references [29, 30].

2.1.3.1. Downlink processing flow

The downlink processing is relatively straight forward. The base station transmits a set of downlink physical channels, typically one for each terminal, plus a set of common channels. All the common channels are transmitted to all users within the reach of the base station. Fig. 2 shows the data flow for downlink user data processing. As explained before, incoming data from the media access control (MAC) layer includes two different streams. One is the DPDCH for data and the other is the DPCCH. Cyclic Redundancy Check (CRC) and forward error correction coding is firstly added to the streams. These are then sent through a rate matcher that assures that the data rate of the stream matches the requirements of the physical layer. The stream is then interleaved, segmented into slots, and interleaved again. Finally, the stream is mapped and spread to the chip rate and output to the radio frontend.

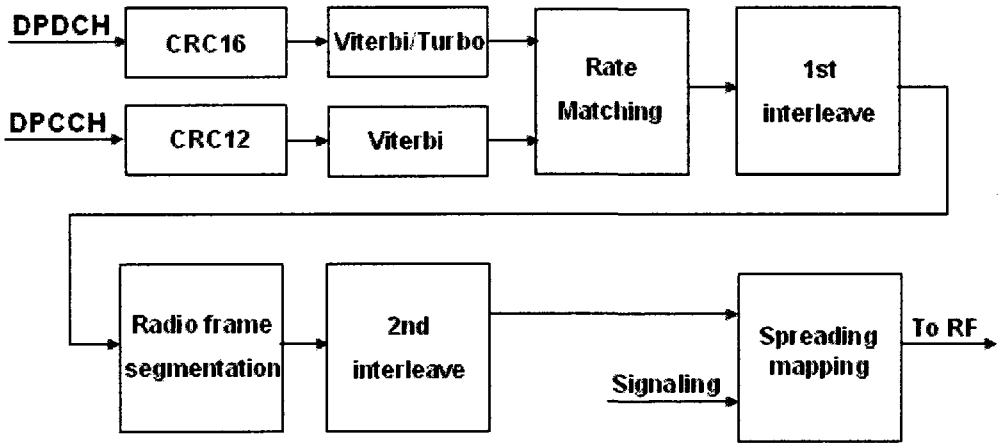

Figure 2 Downlink transmission flow [29]

2.1.3.2. Uplink processing flow

The uplink processing flow is similar to the downlink flow but involves a much higher computation load. Fig. 3 shows the processing flow for the uplink reception. First in the flow is the multipath combination, which is based on a multipath search filter and a Rake receiver working in cooperation. Since each terminal will experience different

multipath propagation conditions, this will require the use of one multipath searcher and combiner per terminal in the base station. The Rake receiver will sum the multiple paths and de-spread the incoming signal. The recovered stream will then be sent through deinterleaving, followed by reverse rate matching. It is then sent through radio frame reassembly and a second deinterleaver before the forward-error-correction decoding is used to restore the received data, which is then sent to the MAC layer.

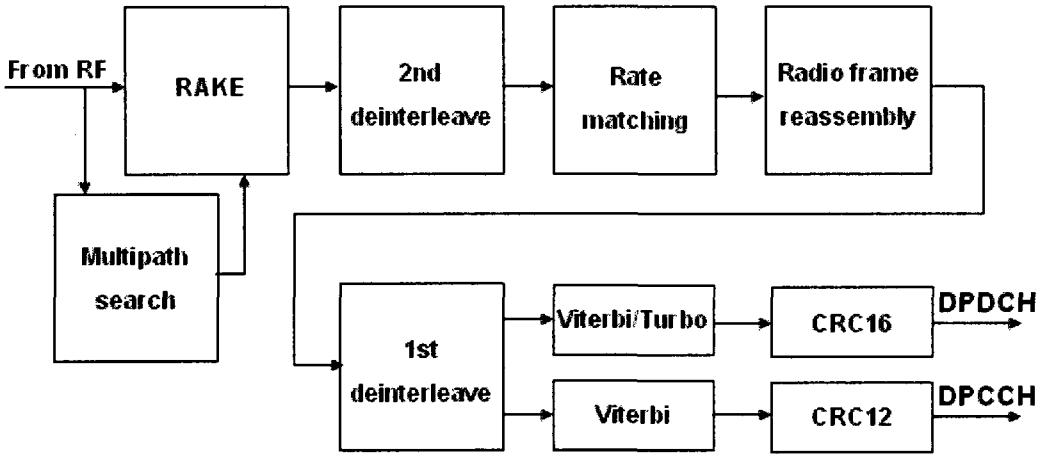

Figure 3 Uplink transmission flow [29]

2.1.4. Channel Coding

The main purpose of channel coding is to introduce redundancy into the transmitted data to improve the wireless link performance. Channel codes can be used to detect as well as correct errors. The WCDMA systems have provision for both error detection and error correction. The channel coding scheme in a WCDMA system is a combination of error detection, error correction, along with rate matching, interleaving, and transport channels mapping onto/splitting from physical channels [30]. Error detection is provided by a Cyclic Redundancy Check (CRC) code while there are two alternatives for error correction schemes specified for the WCDMA. The error correction schemes are convolutional coding and Turbo coding. For standard services that require BER up to10^{-3}, which is the case for voice applications, convolutional

coding is applied. For high-quality services that require BER from 10^{-3} to10^{-6}, Turbo coding is used.

2.1.4.1. Turbo coder

A Turbo coder uses a Parallel Concatenated Convolutional Code (PCCC) which is implemented with two identical 8-state Recursive Systematic Convolutional (RSC) encoders and with an interleaver [28, 30]. In Fig. 4, the structure of a Turbo encoder with coding rate of 1/3 is illustrated. As specified by the UMTS standard, the Turbo decoder can encode a data block with a maximum of 5114 bits. The number of bits in a data block to be encoded is represented by the parameter *K*.

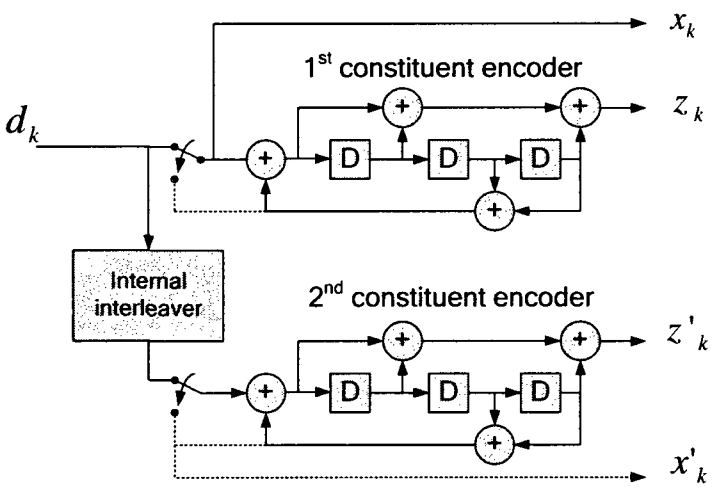

Figure 4 Turbo encoder structure [30]

The output of the encoder includes three binary streams, namely *x*, *z* and *z'*, which are multiplexed in time. The resulting stream is organized according to equation 1. The *x* stream (or systematic code) corresponds to the original input bit sequence *d* which is associated to the *k* index. The *z* and *z'* streams represent the parities generated by the first and second RSC respectively. The input sequence of the second RSC is block-interleaved to ensure coding diversity.

$$\textit{Encoder output} = x_1, z_1, z'_1, x_2, z_2, z'_2 ..., x_K, z_K, z'_K \quad (1)$$

Because the convolutional decoder algorithms require the knowledge of the initial and final states of the encoder, additional bits must be attached to the message so as to recover the transmitted information. This technique is known as Trellis termination and permits the decoder to benefit from past and future indications. The initial and final states of the RSCs before and after the processing of a block of data are zero. In the Fig. 4 example, three zero-valued bits are shifted to each RSC to converge towards state zero (the switches are in the lower position). Also, the input and output bit termination streams of both RSCs must be attached to the end of the encoded information (equation 1) in accordance to equation 2.

$$\textit{Termination stream} =$$

$$x_{K+1}, z_{K+1}, x_{K+2}, z_{K+2}, x_{K+3}, z_{K+3}, x'_{K+1}, z'_{K+1}, x'_{K+2}, z'_{K+2}, x'_{K+3}, z'_{K+3} \quad (2)$$

Each bit of the encoded information, defined by equations 1 and 2, must be formatted into symbols before transmission through the channel. This formatting, known as mapping, consists in the transformation of a logical level 1 and 0 into a value of +1 and -1 respectively. The resulting encoded symbols have the notation defined by equation 3.

$$\textit{Encoded symbols} = u_1, c_1, c'_1, u_2, c_2, c'_2 ..., u_K, c_K, c'_K \quad (3)$$

Where:

u_k : symbol mapping of x_k, the systematic binary sequence

c_k : symbol mapping of z_k, the parity sequences generated by the RSC1

c'_k : symbol mapping of z'_k, the parity sequences generated by the RSC2

2.1.4.2. Turbo decoder

Fig. 5 illustrates the classic iterative structure based on the Maximum-A-Posteriori algorithm (MAP) of a Turbo decoder. The decoder is made of two MAP decoders, namely MAP1 and MAP2, which are dedicated to the processing of the received parities (*c* and *c'*). Each MAP decoder has three soft-quantized inputs, which are the systematic transmitted symbol *u*, the parity *c* or *c'*, and the *a priori* reliability information attached to the transmitted symbol *u* at time *k*.

The MAP decoder has a single soft output: the *a posteriori*, or refined, reliability information. This reliability information is known as the extrinsic Log-Likelihood-Ratio (LLR). As for the encoder, the function of the interleaver/deinterleaver, I / I^{-1}, is used to introduce statistic diversity between symbols in the *c* and *c'* sequences. The properties of the interleavers considerably affect the Bit Error Rate (BER) performance of the Turbo decoder [2, 23].

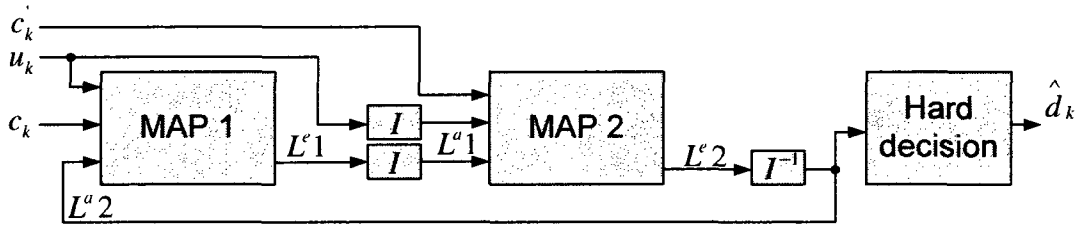

Figure 5 Generic Turbo decoder architecture [4]

A received block of symbols is first demultiplexed to reconstruct both the systematic and the parity sequences (*u*, *c* and *c'*). The MAP 1 decoder generates the *a posteriori* LLR, L^e1, based on the *u* and *c* sequences and the *a priori* LLR (the deinterleaved version of L^a2). The L^a2 are initialized to zero because they are not defined for the first iteration. The MAP 2 decoder then evaluates the associated *a posteriori* LLR, L^e2, using the *c'* parity and the interleaved version of *u* and L^a1. These two consecutives MAP processing steps constitute an iteration of the Turbo decoding. A

Turbo decoder typically requires between 4 and 8 complete iterations to converge and to provide a reliable LLR. The estimation of each bit d is recovered by extracting the sign of each element of the LLR sequence L^e2.

The needed number of decoding iterations strongly depends on the channel's conditions. Furthermore, if the received symbols are distorted, additional iterations are necessary to achieve convergence of the LLR. Because a single iteration involves a large amount of processing, various strategies have been developed to terminate the decoding procedure as soon as the LLR is reliable. Common termination criteria include a threshold based on the LLR energy after a minimum number of iterations or a comparison of the MAP decoder 1 and 2 decoded LLR between iterations [5].

2.2. The Vocallo architecture [21]

Very high-performance multi-core Digital Signal Processors (DSPs) are increasingly used for telecommunication equipments to process voice, video, and radio signals. The current generation of multi-core DSPs has enhanced processing capabilities compared to the previous ones. These new DSPs can replace solutions that previously could only be implemented using dedicated ASICs or DSP-ASIC combinations. Since DSPs are software-driven, when sufficiently powerful, they provide more flexibility than dedicated ASICs.

In this section, we describe briefly the architecture of the Vocallo multi-DSP platform which is provided by Octasic Semiconductor Company. Vocallo is a multiprocessor solution which is used to implement the voice, video, and data over IP applications. This solution is delivered in a 15 core, 1.5GHz, low power DSP, based on Octasic's Opus core architecture. Vocallo has a modular and packet-based software architecture which allows designers to write their own software to extend or replace the original software provided by Octasic. The Opus platform, on which Vocallo is built, includes an integrated development environment (IDE) consisting of a standard C compiler, a visual editing environment, a profiler and a debugger adjusted for multi-core

DSPs. The Opus kernel ensures scheduling of modules in a protected-memory and property secure environment for blending modules provided by Octasic with user modules. The Vocallo solution permits several interfaces to match user requirements. The interfaces such as MII, RMII, GMII, UTOPIA, TDM, and HDLC are supported in Vocallo.

The Opus instruction set includes both traditional DSP functions as well as header processing and task management code. For example, in Vocallo, Opus includes optimized voice and video instructions to support media applications. Also, Opus is implemented with a clock-less architecture to attain new levels of performance per watt. The block diagram representing the Vocallo architecture is presented in Fig. 6. The Opus architecture includes a distributed kernel that abstracts processes from cores. Since it is a homogenous 15-core device, functions are not bound to a specific core or HW functionality. Thus, the total performance of the device can be focused on the required feature set. Also, the kernel allows application-controlled core affinity to minimize memory movement.

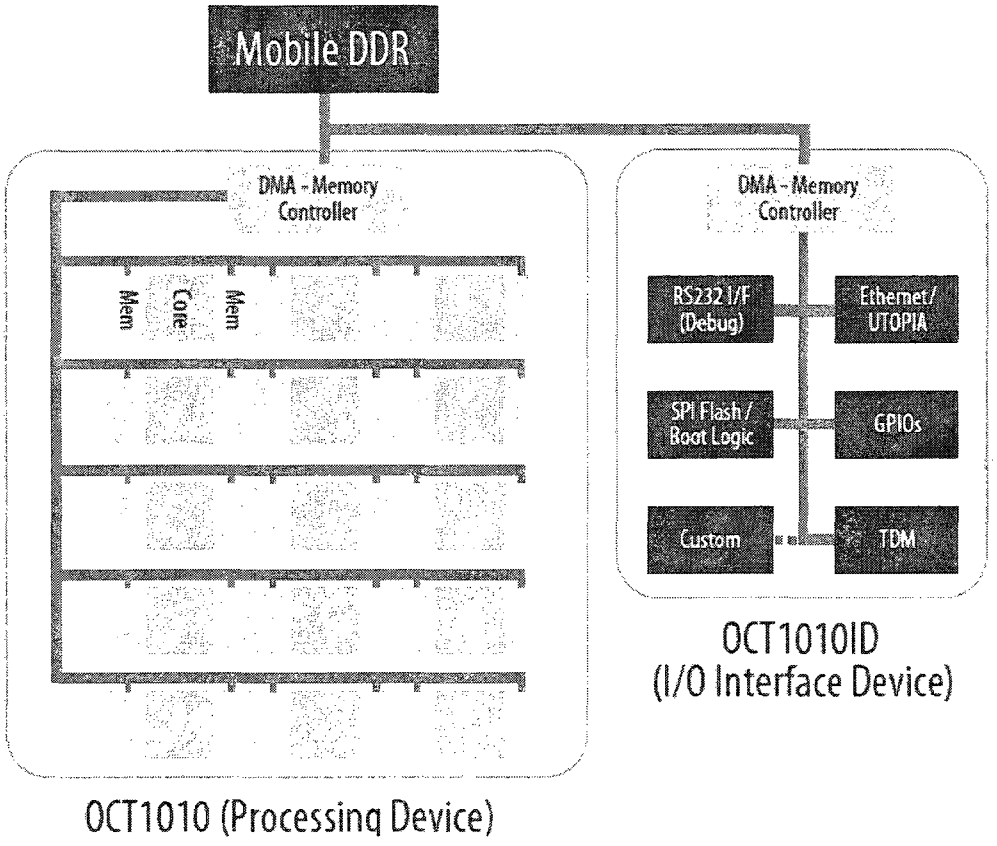

Figure 6 Block diagram of Vocallo architecture [21]

2.2.1. Internal architecture

As previously explained, the Octasic Vocallo device includes an array of DSP cores that are referred to as Opus cores. As shown in Fig. 6, the cores are organized in a 3x5 array and each Opus core includes a 96Kbyte cache memory coupled with a DMA device. The DMA performs direct access between any core and the external memory. When needed, it is possible to disable the cores in the center column, in which case, their cache memory is shared equally between their left and right neighbours (thus adding 48Kbytes of cache to each of them).

The Opus processor core is a Digital Signal Processor that supports both fixed-point and floating-point instructions. Such DSP includes the following elements:

- 96 Kbytes cache memory
- 16 identical Arithmetic Logic Units (ALUs) which operate in parallel
- 64 general-purpose register banks (32-bit registers)
- Extended ALU subsystem for specialized instructions

All logic and arithmetic operations are performed using the data registers. The cache memory contains both, the data and program memory of the processor.

2.2.2. External architecture

The external interface to the mobile DDR memory has the capacity of 32 bits at 166 MHz. The DDR is shared with the cache memories of the Opus cores and also the Input/Output (I/O) interfaces. All the external I/Os are provided by an FPGA.

2.2.3. Power and Performance Optimizations

Flexible, adaptable, and field programmable DSPs used in access and infrastructure equipments typically consume more power and area than their less nimble counterparts such as dedicated ASICs or DSP-ASIC combinations. In modern chips, power dissipation is caused by both static and dynamic phenomena (leakage and switching operations, respectively). A power crisis arises in CMOS technology in 90nm and below. There are various design techniques that can be used to alleviate this power crisis. These techniques vary from very simple to extremely complex and offer a wide range of possibilities for improvement. Some of these techniques are used in Opus and have yielded 4:1 MIPS/power comparative advantage over other leading-edge DSPs in real-life signal processing applications.

2.3. Performance modeling

We intend to study how an application should be scheduled on a network of processors and resources and if such a system meets all the timing requirements. In order to speed up the analysis, we do not want to analyze the specific details about the

application and also the properties of the system resources. We intend to use a model which abstracts away the details on the application and the system resources. Such a model allows us to focus on the timing properties and the resource requirements of the system components. By describing the algorithms and methods to validate the timing constraints of system abstractly, we can better take advantage of their general applicability.

In [15], a modeling and design environment is proposed to allow performance modeling of hardware/software systems. One portion of the design environment consists of a tool for performance modeling of the application (the software) and the architecture it is to execute on (the hardware). In this tool, the computations are characterized by a compute time and the send and receive processes are characterized by an amount of data to be sent. The architecture upon which the application will execute is represented as a set of computational resources and a communications fabric for moving data between them. In [10], an integrated performance based modeling tool (PBMT) is presented. In this tool, the real-time applications are modeled using task graphs specified as control flow and/or data flow. The application model is executed on a variety of SystemC architectures for performance analysis. The task graph of an application model is described using a graphical editor written in java.

A SystemC-based simulation framework is proposed in [13], which enables the evaluation of application-to-platform mappings by means of an executable performance model. In this framework, the application is first represented as a set of untimed reactive SystemC tasks communicating through a unified Transaction Level Modeling (TLM) interface. Next, the processing requirements of each individual task are characterized. Finally, the concept of a Virtual Processing Unit (VPU) is introduced to capture the impact of shared processing elements to the SoC performance. In [25], a framework is proposed, called Virtual Processing Components (VPC), which permits task-accurate performance simulation of applications mapped onto a real-time multi-processor architecture in SystemC. In the VPC framework, the shared hardware resources, like

processors, busses, and memories, are modeled as Virtual Processing Components. Also, each Virtual Processing Component is configured with a scheduling strategy.

2.4. Mapping the system level models into MPSoC platforms

In this section, we describe known methods to combine system level models and architecture level descriptions for MPSoC designs. In [1], a system level design is presented for rapid prototyping of MPSoC starting from a Matlab/Simulink specification. In the mentioned paper, an approach is proposed to create a bridge between the system level specification and the HW/SW architecture at the implementation level. The approach considers system-level specification, multi-level validation, algorithm exploration, refinement, and prototyping generation. The presented design flow combines different languages and tools, such as Simulink, Colif, and Interface generators of Roses, to reach the RTL level. The developed tool fills the gap between Simulink (simulation and validation environment of the applications) and the architectural representation of applications.

Reference [22] proposes an integrated methodology for system design and performance analysis of MPSoC designs. An analytic approach, based on neural networks, is used for high-level software performance estimation which is realized by Matlab software. At the functional level, this analytic tool enables performance evaluation of the considered processors. Thus, the tool refines hardware and software interfaces to provide a bus-functional model. Then, a virtual prototype is generated from the bus-functional model, producing a cycle-accurate simulation model. Such work combines an analytic approach at the functional level and a simulation-based approach at the bus-functional level.

Reference [24] presents a framework for static, analytical, bottom-up temporal and spatial mapping of applications onto MPSoC platforms. The proposed mapping framework permits easy performance evaluation and design space exploration of

heterogeneous systems on chip. Such mapping of applications to a given heterogeneous MPSoC enables not only performance analysis but also refining the system. In the mentioned study, the structure of a framework for automatic mapping is outlined and it is shown that the mapping problem can be treated as a packing problem which can be solved using existing optimization software.

In [9], a software code generation flow based on Simulink is presented to address the problems for software programming on multiprocessor platforms. A functional modeling style is proposed to describe data and control dependent target applications, and a system architecture modeling style is used to transform the functional model into the target architecture. At the system architecture modeling style, the functional model is partitioned into a set of multiple communicating threads on multiple CPU subsystems, which corresponds to the given target architecture. Both functional and system architecture models are described using Simulink. From the system architecture Simulink model, a code generator produces multithreaded code, including thread and communication primitives to abstract the heterogeneity of the target architecture. Also, the multithread code generator applies dataflow based memory optimization techniques, considering both data and control dependency.

In [17], some challenging issues are studied in design space exploration of efficient Network-on-Chip (NoC) designs, especially on the application mapping and scheduling problems. The research mainly focuses on the static analysis of system designs through design space exploration that exploits efficient techniques to solve the application mapping and scheduling problems. The optimization targets can be the synthesis of application mapping, routing and scheduling, as well as network topology. Also, the optimization objectives could be communication latency, application end-to-end delay, and system power dissipation. Also, runtime system management, by online dynamic analysis and scheduling, is studied.

In [26], a design flow is presented that goes from Simulink models to prototypes of mixed hardware/software implementations of these models. The work includes three parts: (1) transformation of a functional model, given in MATLAB/Simulink, into a synchronous reactive model of computation (SR MoC), (2) an automatic SystemC code generation from Simulink models using the SR MoC, and (3) a semi-automatic prototype generator for heterogeneous hardware/software systems implementing the iteration scheduling for SR models. The SR model of computation complements the modeling front-end of the platform-based design flow. The mentioned paper presents a basic mechanism to implement SR models on heterogeneous platforms that can be integrated to the design flow. A transformation step from Simulink models to SR models permits automatic implementation of mixed hardware/software designs from functional Simulink models.

2.5. Multiprocessor scheduling and synchronization

In this section application implementations on the multiprocessor platforms and some scheduling and synchronization concepts in MPSoC environments are discussed based on the reference [16]. If we consider the target application as a set of related tasks, task assignment in a multiprocessor environment is one of the problems to be studied. Control and data dependencies cause constraints between the tasks, and timing constraints of tasks are usually dependent. Most real-time systems with critical timing constraints are built statically, that is, tasks are partitioned and statically bound to processors. The task assignment problem is concerned with how to partition the system of tasks to be assigned to the processors. Another problem is the interprocessor synchronization problem. A synchronization protocol must be used to guarantee that task precedence constraints on different processors are always satisfied. We consider the cases where a multiprocessor system is tightly coupled so that global status and loading information on all processors can be updated at a low cost. The system may include a central dispatcher/scheduler. When each processor has its own scheduler, the decisions and actions of the schedulers of all the processors must be coherent.

2.5.1. End-to-End system functions

The target application may include several system functions where each function can be presented by a set of related tasks. The tasks are described by their release time and deadline, which are independent except for resource conflicts. The tasks in each system function may have some precedence constraints. Here, it is supposed that the precedence graph of each system function is a chain. This simplifies the discussion and covers a wide range of practical situations. As an example, Fig.7 shows a system function including a chain of *m* tasks. The system functions have arbitrary release times and deadlines, and some functions have hard deadlines. Meeting hard deadlines is always considered the primary objective.

The timing constraints extracted from the high-level requirements of the applications are end-to-end in nature. They determine the release time and deadline of each function as a whole. Formally, the release time of a system function is considered as the release time of the first task of the function. Also, the deadline of the function is the deadline of its last task. It is not important when the other tasks of the function complete as long as the last task completes by the function's deadline. The execution of these tasks is constrained only by the dependencies between them and by the fact that they must complete sufficiently early to allow the on-time completion of the last task. Since the timing constraints of such a function are imposed on the tasks at the two ends of the function, they are called end-to-end release time and end-to-end deadline. A function that has an end-to-end release time and end-to-end deadline is an end-to-end function.

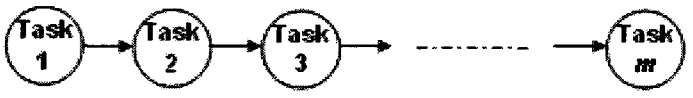

Figure 7 Example of a system function

An end-to-end function in a multiprocessor system may be periodic. An end-to-end function is periodic with period p_i if a chain of $m(i)$ tasks is released every p_i units of time and the tasks in the chain execute in turn on processors based on the tasks assignments. The first task of the end-to-end periodic function is a periodic task with period p_i. The other tasks of the function may or may not be periodic, depending on the method used to synchronize the tasks on different processors.

2.5.2. Elements of scheduling algorithms for end-to end periodic functions

We now present some elements of scheduling the end-to-end periodic functions in a multiprocessor environment. We suppose that each end-to-end periodic function requires resources of more than one processor and that each function includes a chain of tasks which execute in sequence on different processors. Also, each task needs only resources local to the processor on which the task executes. Specifically, a system is now considered which includes n processors, P_j for j = 1, 2, ..., n, and m periodic functions, F_i for i = 1, 2, ..., m. Each function F_i has $m(i)$ tasks, $T_{i,k}$, for k= 1, 2, ..., $m(i)$. These tasks execute in turn on different processors according to the vector c_i of function F_i, $c_i = (V_{i,1}, V_{i,2}, \ldots, V_{i,m(i)})$ where $V_{i,k} = P_j$ indicates that the k^{th} task $T_{i,k}$ executes on processor P_j. Each vector c is called also the visit sequence of the corresponding function F. The mentioned example is shown in Fig. 8.

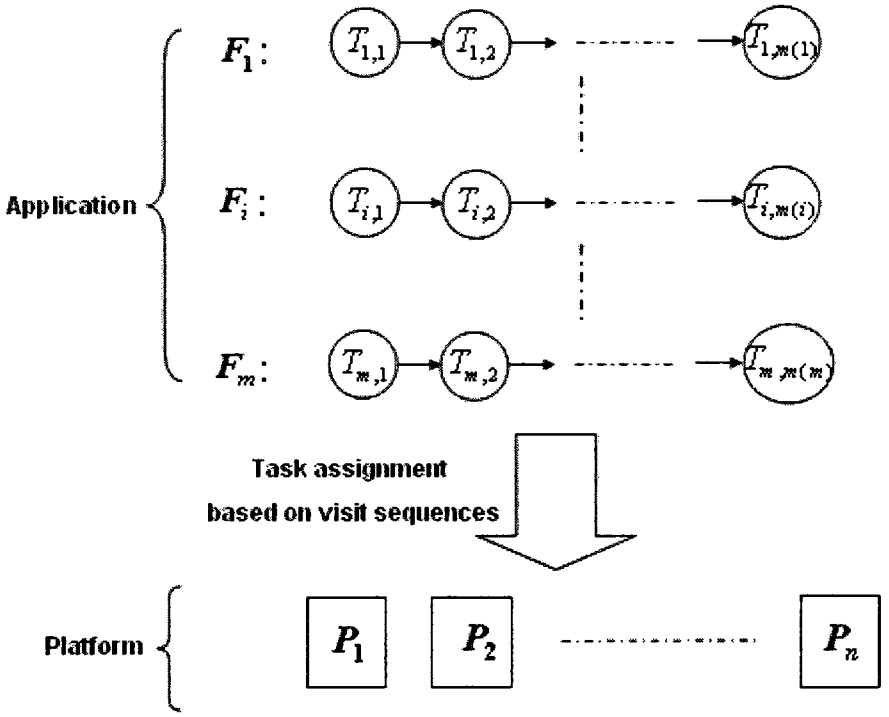

Figure 8 Example of a system including m functions and n processors

The two basic components of any end-to-end scheduling scheme are (1) protocol(s) for synchronizing the execution of tasks on different processors in such a way that precedence constraints among tasks are maintained and (2) algorithms for scheduling tasks on each processor.

2.5.3. Interprocessor synchronization protocols

A protocol that governs when the schedulers on different processors release their assigned tasks is called an interprocessor synchronization protocol. A synchronization protocol never permits the violation of any precedence constraint among the tasks. There are different types of synchronization protocols such as greedy and non-greedy protocols. With a greedy protocol, each task is released on its corresponding processor, as soon as its immediate predecessor task completes. According to a non-greedy synchronization protocol, the scheduler may delay the releases of tasks. The objective is

not only to guarantee that precedence constraints are met but also to shape the release-time pattern of every successor task so that a task behaves like a periodic task if it belongs to a periodic function. One approach in the non-greedy synchronization protocol is to make all tasks periodic. Different kind of non-greedy protocols have been implemented such as the phase-modification protocol, the modified phase-modification protocol, and the release-guard protocol.

2.5.4. Scheduling the tasks on one processor

In this section, the uniprocessor scheduling algorithms for periodic tasks are discussed. There are some well-known priority-driven algorithms for scheduling periodic tasks on a processor which are introduced here. A simplifying assumption is made that the assigned tasks on the processor are independent.

A priority-driven scheduler (a scheduler that schedules tasks according to some priority-driven algorithm) is an on-line scheduler. It assigns priorities to tasks after they are released and places the tasks in a ready task queue in priority order. At each scheduling decision time, the scheduler updates the ready task queue and then schedules and executes the task at the head of the queue. The priority-driven algorithms for scheduling periodic tasks are classified into two types: fixed priority and dynamic priority. A fixed-priority algorithm assigns the same priority to all instances of each task. In other words, the priority of each periodic task is fixed compared to the other tasks. In contrast, a dynamic-priority algorithm assigns different priorities to different instances of each task. Thus, the priority of each instance of task with respect to that of the other tasks can change.

A well-known fixed-priority algorithm is the rate-monotonic (RM) algorithm. This algorithm assigns priorities to tasks based on their periods: the shorter the period, the higher the priority. A well-known dynamic-priority algorithm is the Earliest-Deadline-First (EDF) algorithm which assigns priorities to individual instances of a task based on their absolute deadlines. A criterion which is used to estimate the performance

of algorithms used to schedule periodic tasks is the schedulable utilization. The schedulable utilization of a scheduling algorithm is defined as follows.

A scheduling algorithm can feasibly schedule any set of periodic tasks on a processor if the total utilization of the tasks is equal to or less than the schedulable utilization of the algorithm. Clearly, the higher the schedulable utilization of an algorithm is, the better the algorithm. The ratio $u_i = e_i / p_i$ is called the utilization of a periodic task T_i where e_i and p_i are respectively the maximum estimated execution time and the period of the task. u_i is equal to the fraction of the time a periodic task with period p_i and execution time e_i uses the processor. The total utilization U of all the tasks on the processor is the sum of the utilizations of the individual tasks in it. Since no algorithm can feasibly schedule a set of tasks with a total utilization greater than 1, an algorithm whose schedulable utilization is equal to 1 is an optimal algorithm which provides the total utilization of the resource.

2.6. Worst Case Execution Time (WCET) based design

One critical subject in the real-time systems is to insure that the timing deadlines for providing the system responses are respected besides providing the logically correct results. Therefore one delicate design problem in such systems will be the variability of the tasks execution time. Considering such variability, most scheduling algorithms used in uniprocessor or multiprocessor real-time systems are based on the worst case execution time (WCET) of tasks [12]. Due to such variability, most scheduling algorithms used in uniprocessor or multiprocessor real-time systems are based on the worst case execution time (WCET) of tasks [12]. The WCET of tasks are constant, consequently the real-time system models become deterministic, thus easier to understand and implement. All analytical methods used by real-time application designers produce only estimates of the WCETs. The actual execution times usually remain unknown, until the tasks complete their execution. The problem of such WCET based design is that, in real-time applications with a significant variability of the

execution time, the scheduling analysis based on WCETs leads to low processor utilization [12]. Considering the significant variability existing in the processing effort of the target application and in order to improve the resources utilization, in this project, we use more precise execution time estimates for scheduling the application instead of WCET estimates.

2.7. Scheduling flexible applications

In this section, we describe some flexible computation techniques based on [16]. The term flexible computation (application) refers to a wide class of applications that are designed and implemented to trade off, at run-time, the quality of the results (services) they produce for the amount of time and resources they use. In particular, a flexible application can reduce its time and resources demands at the expense of the quality of its results. As long as the user finds its result quality acceptable, a flexible application can degrade gracefully when resources are limited and the demands of competing workloads increase. There are different methods to implement flexible applications with firm quality.

One way to make an application adaptable to change in resource availability and competing demands is to structure each task so that it has an *optional* component. The optional component execution is not necessary for the task to produce an acceptable result. In contrast, the part of the task that must be completed in time is called *mandatory*. When there are enough resources, the optional component is also executed before the task's deadline. If the optional component, or a portion of the optional component, is not completed, the result quality of the task degrades. A result with an acceptable but degraded quality is an imprecise result. Depending on the characteristics of flexible applications, different implementation methods such as *sieve*, *milestone* and *multiple versions* can be used. For example, in the multiple-version method, each flexible task is considered to have a primary version and one or more alternative version(s). Algorithms for scheduling flexible applications have two objectives. The first

is to ensure that each task will produce an acceptable result on time. The second objective is to maximize the result quality of each flexible application. In references [3, 7 and 14], some scheduling algorithms are proposed to deal with imprecise computation models.

There are many methods to quantify the quality of result of individual tasks, and different quantifications give us different performance criteria. The quality of result of a given task is typically measured in terms of the error in the result. The error in a result is the distance between the result and the desired, precise result. Many algorithms for scheduling flexible applications try to minimize total error, average error, or maximum error of all tasks in the system. These performance measures are called *static quality metrics*. Such static metrics are not applicable where the effects in results produced by periodic tasks are cumulative. For example, in the case of periodic tasks with mandatory and optional components, if the optional component of tasks in a number of consecutive periods is not executed, the optional component of a subsequent task may no longer be optional. Such periodic tasks are called *error-cumulative* tasks. In systems that include *error-cumulative* tasks, the scheduling algorithms use the cumulated error of tasks as the performance metrics.

2.8. Mapping and scheduling of Turbo decoding in MPSoC platforms

In this section, we present some studies about mapping and scheduling of Turbo decoding process on several processors. In [6], a decode apparatus including an array of Turbo decoders is presented to decode a plurality of encoded messages received over a noisy transmission link. An analyzer is considered to indicate the received signals strength and to measure the carrier-to-noise ratio of the signal. Using a look up table, the optimized number of decode iterations for each message is extracted depending on the channel condition. One scheduler means is considered for scheduling demodulated message packets to each of the plurality of decoder processors, depending upon the estimated optimum number of decode operations. Allocation of the message packets to

the plurality of decode processors is made to optimize overall utilization of the decode processors.

In [18], a method is presented for scheduling a decoding process of coded data blocks transmitted over a communication link on several decoders. According to the method, the coded data blocks are stored in a queue if all decoders of iterative parallel decoders are unavailable. When any of the decoders is available, the first coded block of the queue is moved to the decoder. The presented scheduling method improves the resource utilization by automatically adapting the maximum number of decoding iterations in the decoders depending on the bit rate received. In this way, the possibility of supporting high bit rates with a limited number of decoders is provided by applying some degradation of decoding process which is achieved by decreasing the maximum number of decoding iterations.

In [8], [19], and [20], a multiprocessor based Turbo decoder implementation is investigated. In these investigations, the Turbo decoder algorithm is considered as a set of parallel parts which are mapped to several processors. The implemented algorithm parts have data dependencies, which impose data communications between the processors. These studies propose hardware architectures adapted to the parallelization of the Turbo decoder algorithm for reducing the communication latency between the processors during execution.

2.9. Conclusion

In this chapter, we presented some basic concepts and literature reviews relating to our project where its contents will be referred during the next chapters of this thesis. Firstly, we described the UMTS base-station and the WCDMA processing. Thus the downlink and uplink WCDMA processing flow which should be applied to the respectively transmitted and received data blocks in a UMTS base-station were explained. Afterward we introduced the channel coding in the wireless telecommunication systems and specifically the Turbo coding/decoding was described

which was utilized to provide high-quality services. In this way, we presented a description of the target application of this project.

Then we described the architecture of the Vocallo multi-DSP platform which is used in this project. Afterward we presented a literature review of the references [10, 13, 15 and 25] which described some developed methods of performance modeling. All the mentioned references included the concepts such as tasks to represent different parts of a target application, some temporal parameters which were assigned to the tasks and also individual units to represent different parts of the target platform. In Chapter 3, we will present that our proposed methodology of performance modeling utilizes the similar concepts mentioned in [10, 13, 15 and 25] but with using different development methodology.

Also we presented a literature view of some references on mapping the system level models into MPSoC platforms which was the subject of the main OPERA project. The mentioned references described the methods to combine system level models and architecture level descriptions for MPSoC designs. Afterward we presented some basic concepts on the multiprocessor scheduling and synchronization. For this purpose, we introduced the end-to-end system functions and the elements of scheduling algorithms for end-to-end periodic functions where each function was represented as a chain of tasks. Thus, we introduced some interprocessor synchronization protocols and also the methods for scheduling the tasks on each processor. These presented concepts will be referred in Chapter 4 to explain our proposed method of mapping and scheduling the uplink WCDMA processing on the MPSoC platform.

Then we explained the WCET based design and the advantage and disadvantage of such method in a real-time system implementation was described. Afterward we presented some methods for scheduling the flexible applications which allowed to trade off the quality of results for the amount of processing time and the required resources. Our proposed flexible scheduling methods in Chapter 4 will be inspired from these presented methods for scheduling of flexible applications. Afterwards we presented a literature review of the references [6, 8, 18 19 and 20] on the mapping and scheduling of

Turbo decoding in an MPSoC platform. In Chapter 4, we will propose some methods for scheduling the Turbo decoding processes on an MPSoC platform which provide much more flexible degradation of the decoding and consequently more utilization of the resources compared to [6, 18]. Also unlike to [8, 19 and 20], we will consider that the Turbo decoder algorithm consists of only one monolithic task and the Turbo decoding process on each coded block is performed totally on one processor. Thus the Turbo decoding process on the individual processors will be data independent which reduces the data communication between the processors.

CHAPTER 3

PERFORMANCE MODELING

As mentioned earlier, we strongly need to devise a performance model for estimating the execution time of applications on the platform before their implementation. For this reason, we propose a performance modelling methodology based on a Matlab model which includes the application execution timing characteristics and the high level primitives of the platform. It is assumed that the execution timing characteristics of different parts of an application have been extracted before (using, for example, the simulator of the target platform) and are introduced in the performance model. Simulating such Matlab model emulates the execution time of different parts of the application which gives the final execution performance without requiring any functionality examination. In this way, the model provides rapid estimation of execution performance which allows validating the different strategies for mapping and scheduling of applications on the platform. Also, the model makes it possible to estimate the capacity of the platform to support the requested traffics while providing the statistical analysis on the services provided by the system.

In this chapter, we first describe the proposed performance modelling method which is divided into three steps. After that, we describe the functionalities of different parts of the proposed performance model in more details. Then, a model example is presented and some simulation results for that example are described. These simulation results are used to provide more explanations on estimating the timing parameters in a performance model. A conclusion is provided for the presented concepts in this chapter.

3.1. Steps to create a performance model

The proposed performance modelling methodology includes three steps of modelling to create a final performance model of a given application and platform. At each modelling step, some software and hardware elements of the system are modelled

which allow developing a complete model of the system. In the first step, we model the application which needs to be mapped on the target platform. The application must be partitioned into several processing segments and these segments should be assigned to different processors of the platform. Thus, in the first step of our modelling process, all the processing segments are described as individual tasks and each task includes timing parameters regarding the execution of the corresponding processing segment on the target processor. Also, the data transmission between the processors due to the partitioning is modelled by individual tasks that include timing parameters describing the corresponding data transmission. In this way, depending on the mapping strategy used to assign the processing segments to the processors, we create a model of the mapped application that includes several tasks.

In the second step, we model the architecture of the target platform as a set of units to represent different processors and also the communication part of the platform. Afterward, depending on the mapping strategy, the tasks created in the first step are assigned to these created units. We should mention that the tasks that represent the data transmission between the processors are assigned to the unit that represents the communication part of the platform. At the end of the second step, the created model includes several units that represent the architecture of the platform where each unit includes several tasks that represent the timing parameters of the processing or data transmission. When the model is simulated, tasks on different units should be executed based on an order which is forced by the precedence relations between the modelled processing segments. A task is considered ready to be executed when execution of the preceding tasks on the corresponding task chain is finished.

At the third modelling step, we consider another unit in the model which is responsible for managing task execution on different units. This supplementary unit is called Master and it models another processor of the platform. We suppose that the Master includes a module called Synchronizer that analyzes when the different tasks are ready for execution by verifying the executing situation of all tasks. Also, the Master

includes a module to define the simulation time. Such provided simulation time is utilized as a reference to define the timing parameters of the tasks during the simulation. The other units which include the tasks are called Slaves and they are responsible of executing their corresponding tasks. We suppose that each Slave includes a module called Scheduler in addition to its corresponding tasks. The Scheduler is responsible for selecting one task between the tasks of the Slave which are ready for execution based on its scheduling strategy. The task selected by the scheduler is executed by the Slave.

In this way, a performance model is created which represent a high-level mixed description of the target application and the platform architecture. During simulation of such a model, all the mentioned Master and Slave units are simulated. We consider that the simulation is done in several simulation stages. At each simulation stage, first, the Master unit is simulated then the Slaves are simulated one by one. Simulating the Master unit provides the times when the tasks are ready for execution and the simulation time at the current stage of simulation is determined. Simulating the Slave units may lead to executing some ready tasks and to updating several task timing parameters. In the following, more details are provided about the mentioned modelling steps.

3.1.1. Modelling the Mapped Application on the Multi-Processor Platform (Modelling Step 1)

We suppose that different parts of the target application are mapped on a multiprocessor platform based on a given mapping strategy. In order to create the performance model, we first model the application which needs to be mapped on the platform. We suppose that the target application includes a set of processing segments that should be mapped to different processors of the platform. Also, we know that, after mapping and implementing the application, some data transmission may need to be performed between the processors to transmit the data between the dependent processing segments executed by different processors. In order to demonstrate the concepts, we consider a simple example of a mapped application which is presented in Fig. 9. This example corresponds to an application composed of four processing segments that are

mapped on the processors number 1, 2, 4, and 5 of the target platform. The data flow in the application and in the platform for this example is presented in Fig. 9.

In order to model the mapped application, each processing segment assigned to a processor is modelled as an individual task. Also, each data transmission that must be done between two processors or one processor and the input/output subsystem of the platform is modelled as a task. Each task includes timing parameters related to the execution of the corresponding processing segment on a target processor or the data transmission between two parts of the platform. Such a modelling process and the model of mapped application for the above mentioned example are also shown in Fig. 9.

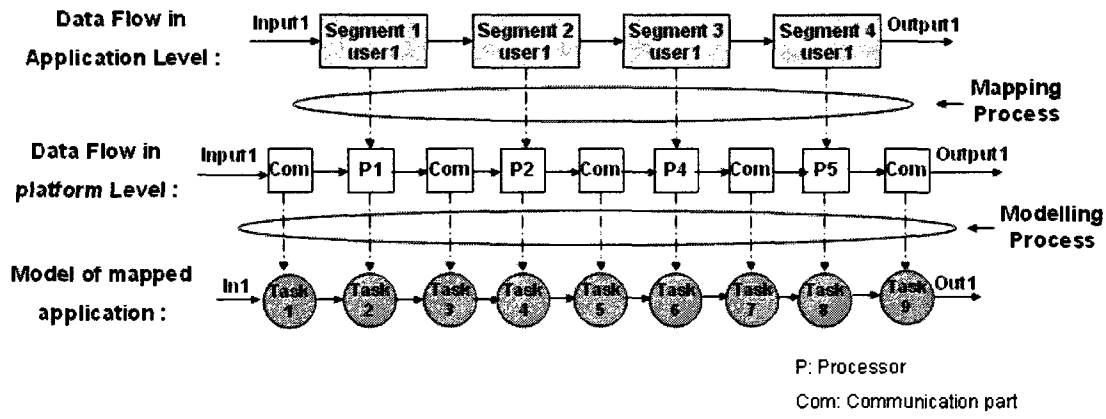

Figure 9 Modelling a mapped application

The tasks that model the mapped application have precedence relations that describe their execution order as shown in Fig. 9. We suppose that the model of the mapped application includes a total of m tasks. In order to describe the execution precedence of tasks, we use a vector called *Dep*. This vector includes m elements where each element denotes the task that precedes, on the task chain, the task corresponding to the position in the vector. Fig. 10 presents an example of the *Dep* vector which shows the vector elements corresponding to the mapped application shown in Fig.9. As an example, the third element of the vector in Fig. 10 which corresponds to task3 is equal to 2. This indicates that task2 immediately precedes task3 in the task chain or, in other

words, task3 directly depends on task2. If a task does not depend on any tasks or it is activated by an input signal, its corresponding element in the vector is set to the value -1. In the dependency configuration described by the *Dep* vector, we suppose that each task depends directly only on one single task or one single input.

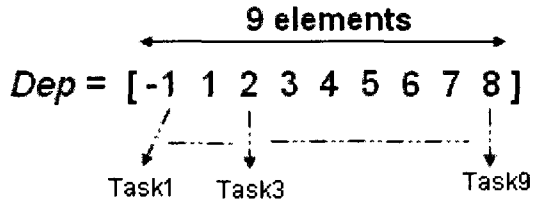

Figure 10 Example of vector *Dep*

3.1.2. Structuring the model of mapped application (Modelling step 2)

In addition to describing the mapped application in the form of a task set, we introduce also the structure of the platform. We model the target platform as a set of units where each unit represents one processor or the communication part of the platform. Each unit is represented by an individual Matlab piece of code. Thus, based on the mapping strategy, we assign the created tasks representing the mapped application to these units. In this way, the performance model includes several units where each unit contains several tasks. Fig. 11 represents a schema of the performance model for the mapped application shown in Fig. 9. As can be seen in this figure, the performance model in this example contains five units where unit1 represents the communication part of the platform and where the other units represent the processors 1, 2, 4 and 5. This model does not include a unit corresponding to the processor number 3 because there is no assigned processing segment to this processor in that example. The tasks representing the data transmission on the platform are assigned to unit1 and the other tasks are assigned to the units representing the corresponding processors.

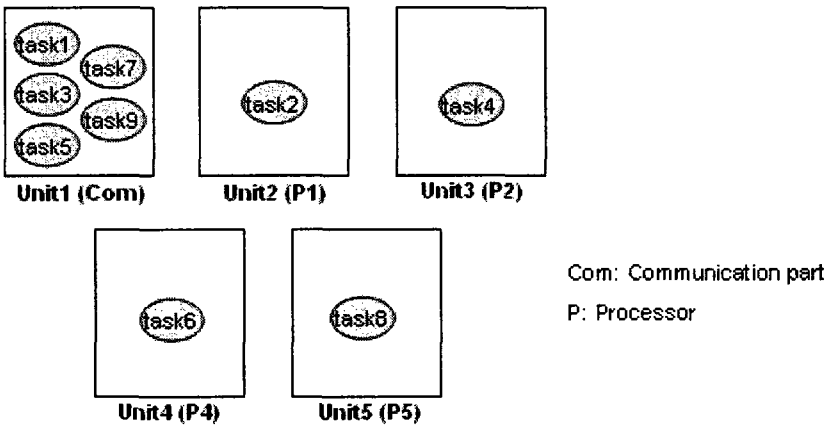

Figure 11 Example of the structured model of a mapped application

Task assignment in the model is described using a vector called *tsk_unit*. The number of elements in the vector *tsk_unit* is equal to the total number of tasks. Each element of this vector indicates the assigned unit of the corresponding task, which represents a processor or the communication part of the platform. Fig. 12 shows an example of the vector *tsk_unit* that presents the elements corresponding to the example in Fig. 11. For example, the first element of *tsk_unit* in Fig. 12 determines that task1 is mapped to unit1, which represents the communication part of the platform, while the fourth element of vector presents that task4 is mapped to the unit 3 representing processor number 2.

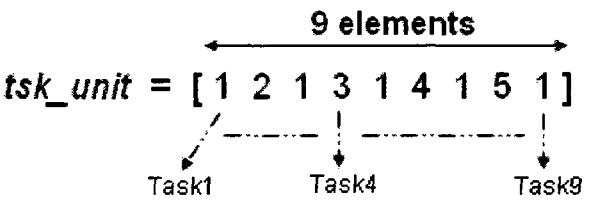

Figure 12 Describing the structured model of mapped application

3.1.3. Creating a Master/Slave Structure (Modelling Step 3)

Up to this point, we created a model that includes several units representing different parts of the platform where each unit contains several tasks. During the simulation of the target model, the tasks representing the mapped application should be executed on different units in a predefined order described by the *Dep* vector. Also, in the model, we consider a module called Synchronizer which verifies the executing situation of all tasks on different units and determines when the tasks are ready for execution, considering the completion time of the previous tasks on the task chain or arriving time of signals on the corresponding inputs. We suppose that the Synchronizer module is placed on one modeled unit called Master. The unit dedicated to the Synchronizer represents one processor of the platform which synchronizes the execution of different application processing segments in the actual implementation. The other units of the model containing the tasks are called Slaves. We consider that each Slave unit includes a scheduler module which determines the task execution order on the Slave. Fig. 13 shows the Master/Slave structured model for the example shown in Fig. 11, which also presents the interconnections between the different elements of the model. In order to generalize the demonstration, the tasks assigned to each Slave in Fig.13, are shown as a task set.

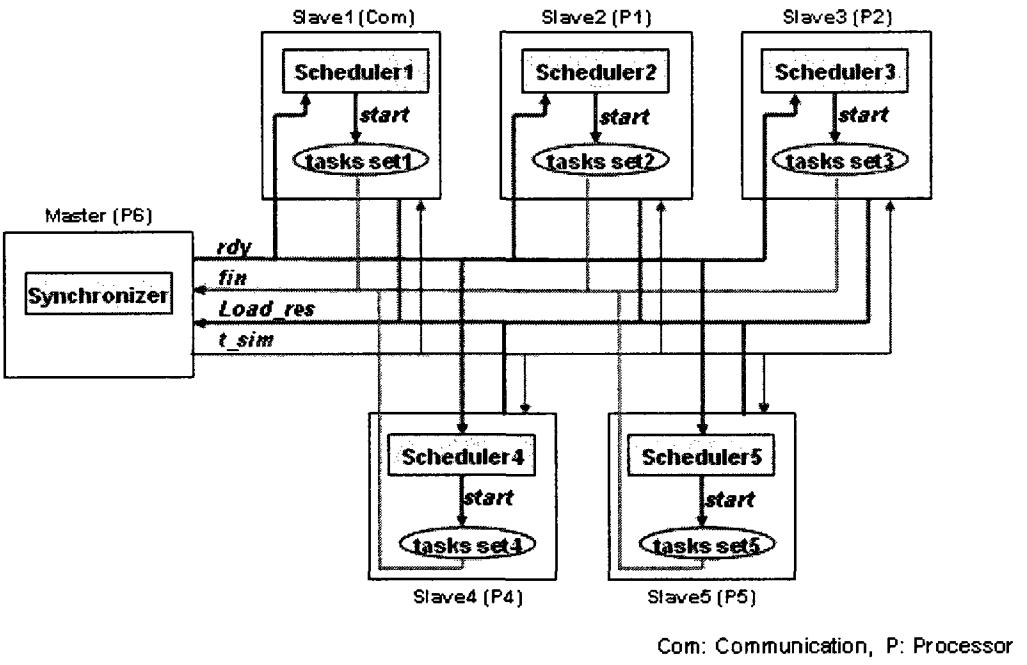

Figure 13 Performance model block diagram for an example case

The descriptions of the shown parameters in Fig. 13 are presented as follows:

Parameters *rdy*, *st*, *start*, and *fin*:

As explained earlier, the application mapped on the platform is modelled as a set of tasks. Each task includes timing parameters related to the execution of one part of application or to data transmission on the platform. Four timing parameters, called *rdy*, *st*, *start*, and *fin*, are associated with each task. *Rdy* is determined by the Synchronizer and represents the time when the task is ready for execution. If a task precedes another task in its task chain, its parameter *rdy* is defined based on the time when the execution of its previous task is finished. Otherwise, if the task is the first task in the task chain and it is activated by an input signal, its parameter *rdy* is defined based on arrival time of the corresponding input signal. *St* represents the earliest time when the task can be executed on the corresponding Slave and it is defined based on the time when the task is ready for execution and the time when the Slave becomes free. *St* is defined by Synchronizer and it is used to define simulation time which will be explained later in details. *Start*

represents the time when the execution of the task begins and is determined by the corresponding Slave. *Fin* represents the time when the execution of the task is completed and is also defined by the corresponding Slave. We introduce four vectors called *rdy*, *st*, *start*, and *fin* where each vector has *m* elements (*m* is equal to the number of tasks). Each element of these four respectively denotes the parameters *rdy*, *st*, *start*, and *fin* for the corresponding task. The values of these vectors are updated during simulation. Since *st* is used internally to the Synchronizer, it is not shown in Fig. 13.

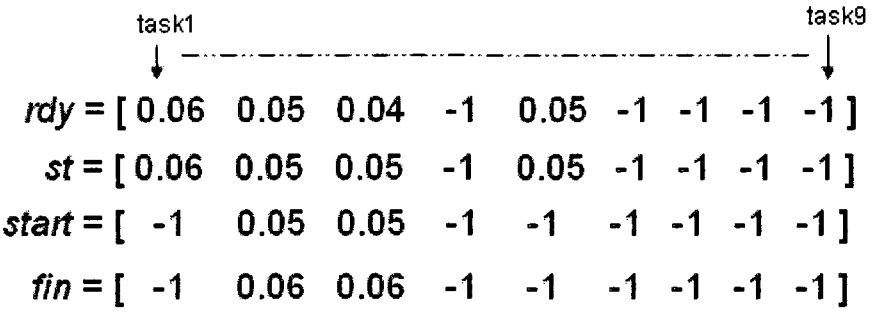

Figure 14 Example for vectors *rdy*, *st*, *start*, and *fin*.

As explained before, the simulation of a performance model is done in several stages and, at each simulation stage, first, the Master unit is simulated then the Slaves are simulated one by one. Fig. 14 presents a sample for the vectors *rdy*, *st*, *start*, and *fin* taken from one given stage of simulation for the example described in the previous sections. The vector *rdy* in this example shows that task1, task2, task3, and task5 are ready for execution at respectively 0.06, 0.05, 0.04, and 0.05s. The parameter *rdy* for the other tasks is equal to -1, which shows that the corresponding tasks are not ready for execution after this one stage of simulation. As shown by the *st* vector, the earliest start time for execution of task1, task2 and task5 is equal to their ready time, which means that the corresponding Slaves of these tasks are not busy at the moment that the tasks become ready. The earliest start time of task3 is equal to 0.05s, which is greater than the ready time of this task and corresponds to the time when the corresponding Slave will be free. Values of -1 in vector *st*, indicate that the corresponding tasks are not ready for execution. The values in the *start* vector indicate that only task2 and task3 are executed

and the execution of both tasks started at 0.05s. Vector *fin* indicates that the execution of task2 and task3 is finished at 0.06s. (In this example, we suppose that the execution time of all tasks is equal to 0.01s) Values of -1 in vectors *start* and *fin* indicate that the execution of the corresponding tasks are respectively not started and not finished.

Table 1 summarizes the definition of the presented timing parameters for a task estimated during each stage of simulation. Also, all the possible cases of the timing parameters during each simulation stage for a task are shown in table 2.

Table 1 Task timing parameter definition.

		Definition
Provided by Master	**rdy**	***rdy*≥0:** *Rdy* indicates the time when the task is ready for execution considering the execution of its previous task in the task chain or arriving time of corresponding input signal. ***rdy*=-1:** Task is not ready for execution.
	st	***st*≥0:** *St* indicates the earliest time when the task can be executed considering its ready time and the time when the corresponding Slave is free. ***st*=-1:** Task is not ready for execution.
Provided by Slave	**start**	***start*≥0:** S*tart* indicates the time when the execution of task is started. ***start*=-1:** Execution of task is not started.
	fin	**fin≥0:** **1-** If the task is executed at current stage of simulation, *fin* indicates the time when the current execution of task is completed. **2-** If task is not executed at current stage of simulation, value of *fin* corresponds to the previous execution of task. It means that this value of completion time has not yet been used to determine the ready time of the next task in the task chain (its dependent task) or the task has not any dependent task. **fin=-1:** Execution of task is not completed (task is not executed at current stage) and the previous execution of task (if there is) has been used to determine the ready time of corresponding dependent task.

Table 2 Possible cases of timing parameters for a task.

		Case1	Case2	Case3	Case4	Case5
Provided by Master	**rdy**	≥0	≥0	≥0	-1	-1
	st	≥rdy	≥rdy	≥rdy	-1	-1
Provided by Slave	**start**	st	-1	-1	-1	-1
	fin	>start	-1	>0	-1	>0
	Description	Task is ready for execution and it is executed.	Task is ready for execution but it is not executed.	Task is ready for execution but it is not executed. (Value of fin corresponds to the previous execution of task.)	Task is not ready for execution so it is not executed.	Task is not ready for execution so it is not executed. (Value of fin corresponds to the previous execution of task.)

t_sim is the reference simulation time which is provided by the Master and is used by the Slaves to calculate the start time of tasks during simulation.

load_unit is a vector of *n* elements where *n* is equal to the number of Slave units. Each element of this vector denotes the completion time of the last executed task on the corresponding Slave at the current simulation stage and is provided by the same Slave unit. In other words, it represents the time from which the Slaves will not be busy and can execute another task. Fig. 15 shows the estimated vector *load_unit* for the same example and the simulation stage presented in Fig. 14. As explained in the example shown in Fig. 14, task2 and task3 are executed at the current simulation stage and their completion time is equal to 0.06s. Therefore, the *load_unit* parameters of the Slaves corresponding to task2 and task3 (respectively Slave2 and Slave1) have been set to 0.06s as shown in Fig. 15. This figure shows also that *load_unit* of Slave3 has been set to 0.05. Since task4, which is the only task placed on Slave3, is not executed at this stage, the value of *load_unit* for Slave3 corresponds to the previous execution of task4. *Load_unit* for Slave4 and Slave5 is equal to 0, which means that no task is yet executed at those Slaves and they have been available from time equal to 0s.

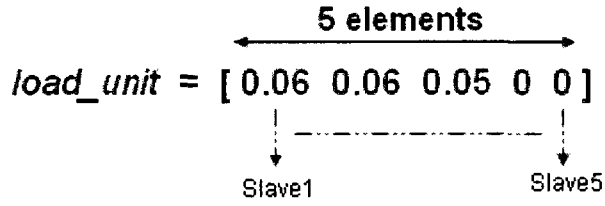

Figure 15 Example of vector load_unit.

3.2. Detailed description of the performance model

Simulating a created performance model is supposed to be done in several simulation stages. At each simulation stage, the Master unit is first simulated then the Slaves are simulated one by one. The model simulation continues until there is no ready task to be executed on any Slave after a given number of received data elements at the inputs. Note that each simulation of the performance model is set to be done for a given total numbers of signals received at the different inputs to the system. In the model, we consider a basic part which describes the order for simulating the Master and Slaves and defines the simulation halting strategy. The pseudo code of this basic part is shown in Fig. 16. In the shown pseudo code, there is a *while* loop where each instance of this loop corresponds to one stage of simulation.

In the following section, we describe the functionalities of different parts of the performance model in more details.

3.2.1. Master

The Master unit includes two modules called Synchronizer and Simulation Time Estimator which are shown in Fig.17. Functionalities of these modules are explained in more details as follows.

Pseudo code of the basic part of model :

```
Executing the configuration file which introduces the vectors and variables used in
the model.

while (1)

   Simulating the master unit

   (Stopping the simulation of model if there is no ready task in the model.)
    if All tasks have been executed after receiving the given total numbers of input
      signals.
      break
    end

   (Simulating the Slave units)
   for i=1 to Number of Slaves
      Simulating Slave i.
   end

end
```

Figure 16 Pseudo code of the basic part of model

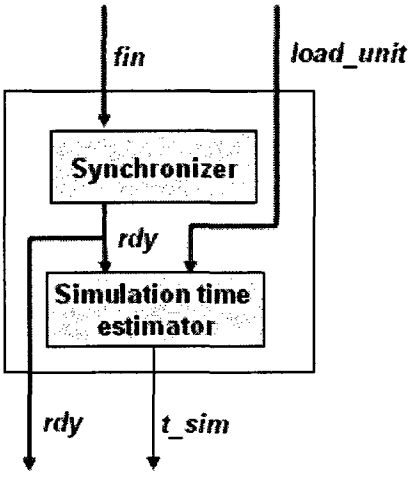

Figure 17 Block diagram of the Master.

The Synchronizer determines the execution ready time of all tasks and updates the *rdy* vector. The Synchronizer firstly determines the ready time of tasks for which their execution depends on other tasks. For this purpose, it uses the precedence relation of tasks described in the *Dep* vector. The flowchart representing the functionality of this

part of the Synchronizer (part1) is shown in Fig 18. As shown in this figure, the Synchronizer verifies the completion time of tasks provided in the previous stage of simulation. If the completion time (parameter *fin*) of a task is greater than zero, the Synchronizer finds the dependent task of the current task. (In order to simplify the explanation, the verified task that has a completion time greater than zero is called the original task.) Thus, the Synchronizer verifies the parameter *rdy* of the dependent task. If the ready time of the dependent task is equal to -1, the value of ready time is set to the completion time of the original task.

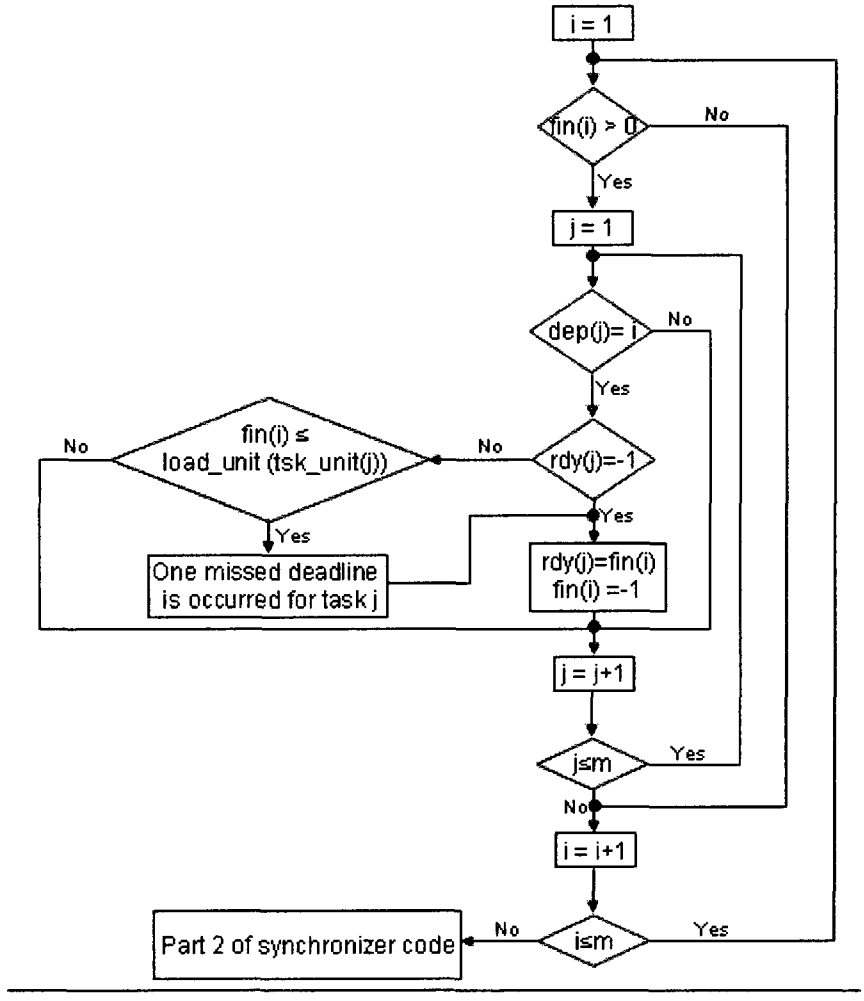

Figure 18 Flow chart of the Synchronizer (Part 1)

If the ready time of the dependent task is not equal to -1, it means that the dependent task became ready at one of the previous simulation stages and that the requested execution has not yet been performed. If the completion time of the original task, which should define the ready time of the dependent task, is less than or equal to the *load_unit* of the corresponding Slave of the dependent task, it means that, when the Slave of the dependent task becomes free, there will be two requested executions for the dependent task. We suppose that there are data buffering limitations and that the input buffer for each task is filled by the next arriving data even if the current data has not been used. Thus, because of such a data buffering limitation, the previously determined ready time of the dependent task is ignored and it is set to the completion time of the original task. Therefore, a missed deadline occurs for the dependent task.

If the ready time of the dependent task is not equal to -1 but the completion time of the original task is greater than *load_unit* of the corresponding Slave of dependent task, it means that, when the Slave of dependent task becomes free, the execution of the original task has not been yet completed. Therefore, the previously determined ready time of dependent task is kept by the Synchronizer and it is not replaced by completion time of the original task.

Afterward, the Synchronizer determines the ready time of tasks which depend on the input signals using the arriving time of data elements at the corresponding inputs. We suppose that the input signals are in the form of data frames which are arrived at the system inputs periodically. The functionality of this part of the Synchronizer (part2) is represented by the flowchart shown in Fig 19. The arrival time of the first frame at each input is set to 0s. As shown in Fig. 19, the Synchronizer first verifies which tasks depend on the input signals. If a task depends on an input signal and if the total number of arrived frames at the corresponding input is less than a specified value, it updates the ready time of the task. For this purpose, it verifies the previously determined value of ready time for the task. If the parameter *rdy* is equal to -1, it means that there is no request for execution of the activated task in the previous simulation stages that has not

been done. Therefore, it increments the number of received frames at the corresponding input (updates the number of current frame), evaluates the arrival time of the current frame and sets the parameter *rdy* to this evaluated time.

If the previously determined value of *rdy* is not equal to -1, it means that there is a request for execution of the task activated in the previous stages of simulation that has not yet been done. In this case, if the arrival time of the next frame at the corresponding input is equal or less than the *load_unit* of the corresponding Slave, the Synchronizer increments the number of received frames at the corresponding input (updates the number of current frame) and sets the parameter *rdy* to the arrival time of the current frame. Because, in this case, when the Slave becomes free (at its *load_unit*), the task has been requested for execution by the arrival of a new frame while the previous request has not been yet done. Considering the data buffering limitation, we suppose that the previously buffered data frame is replaced by the newly arrived frame. Thus the previous request is ignored and a missed deadline has occurred for the task. Such verification for the next arriving frames is repeated and the number of current frame is incremented until the arriving time of the next frame is greater than *load_unit*. This repetition is done to ensure that at the time when the Slave becomes free, *rdy* of the task is set to the arriving time of the last received frame.

If the parameter *rdy* of the task is not equal to -1 but the arriving time of the next frame at corresponding input is greater than the *load_unit* of the corresponding Slave of the task, the Synchronizer does not increment the current frame number and consequently does not update the value of *rdy*. Because in this case, when the corresponding Slave becomes free (at its *load_unit*), there is only the previous request of execution for the task.

The pseudo code describing the functionality of the Synchronizer is shown also in Fig. 20.

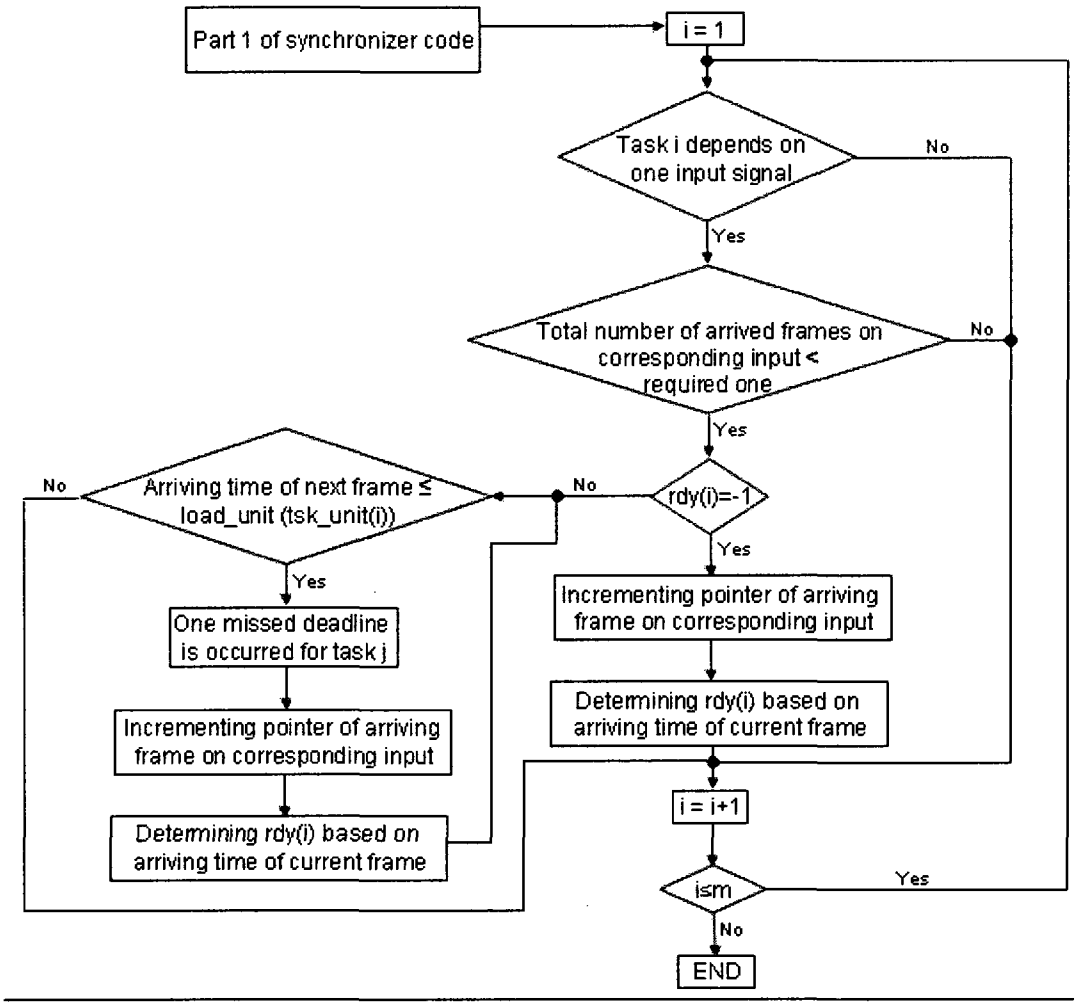

Figure 19 Flow chart of the Synchronizer (Part 2).

The Simulation Time Estimator determines the current simulation time, *t_sim*, at different stages of the simulation. The different Slave units always compare the ready time of their corresponding tasks with this simulation time. If the ready time of a task is greater than this reference time, such ready time is considered to belong to the future and the task is not considered for execution at current stage of simulation. The simulation time is calculated after updating the ready time of the tasks by the Synchronizer.

```
(Determining the ready time of tasks which depend to other tasks)
for i= 1 to m     (m is equal to the number of tasks)
   (Verifying if the completion time of task i has not been used in the previous stage of simulation.)
   if fin(i) > 0
      for j=1 to m
         if ( dep(j) = i )  (Finding the tasks which are dependent to the task i)

            if rdy(j) = -1  (task j with deactivated ready time parameter)
               rdy(j) = fin(i)  (Determining the ready time of the dependent task of task i)
               fin(i) = -1  (Deactivating the completion time of task i)
            else   (task j with activated ready time parameter)
               (Missed deadline case for task j)
               if completion time of task i is equal or less than the loading time of corresponding Slave for
                  task j.
                   rdy(j) = fin(i)  (Determining the ready time of the dependent task of task i)
                   fin(i) = -1   (Deactivating the completion time of task i)
               end if
            end if

         end if
      end for
   end if
end for

(Determining the ready time of tasks which depend on input signals)
for i=1 to m
   if task i is dependent to one input signal and total number of arrived frames at the corresponding
      input is less than a required one.

      if rdy(i) = -1 ( Case of task i with deactivated ready time parameter.)
         Incrementing pointer of arriving frame.

         Determining rdy(i), based on the period and number of current frame arrived at the input signal
          which presents the arrival time of  the related frame.

      else  (Case of task i with activated ready time parameter.)

         (Missed deadline case for task j)
         while ( Arriving time of the next frame is equal or less than the loading time of corresponding
                  Slave)
            Incrementing pointer of arriving frame.

            Ready time of task i is replaced with the arriving time of the current frame. Since the
            previous ready time has not been used, a missed deadline is occurred for the task i.
         end while
      end if
   end if
end for
```

Figure 20 Pseudo code of Synchronizer.

We explain now the method which is used to determine *t_sim*. In order to calculate *t_sim*, the Estimator utilizes the recently calculated ready times of tasks and

also the loading time of the Slaves (*load_unit*). First, it estimates the earliest time that the tasks can be executed on their corresponding Slaves, which are called the earliest start time of tasks (*st*). A vector called *st* with *m* elements is created by the estimator where m is equal to the number of tasks. As explained earlier, each element of *st* represents the earliest start time of the corresponding task. The earliest start time of a ready task is estimated using Equation 4.

$$st\ (j) = \max\ (load\_unit\ (tsk\_unit\ (j)),\ rdy\ (j)) \qquad (4)$$

In Equation 4, *j* and *load_unit* (*tsk_unit* (*j*)) are respectively the task number and the parameter *load_unit* of the corresponding Slave. If the ready time of a task is greater than the time when its Slave becomes free, it means that the corresponding Slave is busy when the task becomes ready for execution and the earliest start time of task is set to the *load_unit* of the Slave. Otherwise, the earliest start time of a task is set to the task's ready time. If a task is not ready for execution, its corresponding *st* is set to -1. Fig. 21 shows two examples for estimating the earliest start time. The considered task in these examples is task 3 which is placed on Slave2.

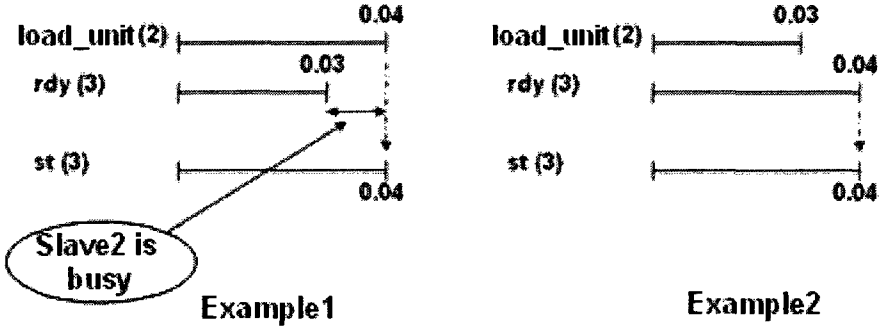

Figure 21 Examples for estimating earliest start time.

In the same manner, all elements of vector *st* corresponding to all tasks are estimated. Then, the minimum value between the earliest start times of tasks is considered as parameter *t_sim*. The pseudo code of the simulation time estimator is presented in Fig. 22. The reason of providing *t_sim* is described in more details later.

```
(Estimating the earliest start time of tasks)
for i=1 to m
   if rdy(i) = -1
      st(i) = -1 (Deactivation of earliest start time of task if the task is not ready)
   else if

      (st(i) is considered as maximum between ready time of task and loading time of
       corresponding Slave)
      if rdy(i)> load_unit(tsk_unit(i) )
         st(i)= rdy(i)
      else if
         st(i)= load_unit(tsk_unit(i) )
      end if

   end if
end for

(Estimating the simulation time)
t_sim=1e5  (Initializing t_sim with a large value)

(Finding the minimum value between the earliest start time of ready tasks)
for i= 1 to m
   if st(i) ~= -1 & st(i)<t_sim
      t_sim = st(i)
   end if
end for
```

Figure 22 Pseudo code of simulation Time Estimator.

3.2.2. Slave

We describe now the functionalities of the Slave units in more details. As explained earlier, a timing reference (*t_sim*) is provided by the Master unit, which represents the simulation time of the current simulation stage, and is used by all Slaves. Note that, if the parameter *load_unit* of a Slave is greater than *t_sim* (simulation time), it means that the corresponding Slave is busy at the current simulation stage and its

functionalities are not evaluated at that stage. The block diagram of a Slave unit is shown in Fig. 23. As shown in this figure, several functionalities are performed by each Slave unit. Firstly, at the Task selection functionality level, a task is selected between the tasks that are placed on the Slave to be executed and consequently the execution start time (*start*) of the selected task is provided. Note that only one task can be selected for execution at a time. Next, the selected task is launched, which provides the execution time (*t_exe*) of the task. Thus, the absolute execution completion time (*fin*) of the task is estimated. Afterward, the loading time of the Slave (*load_unit*) is provided. In the following, these functionalities are described in more details.

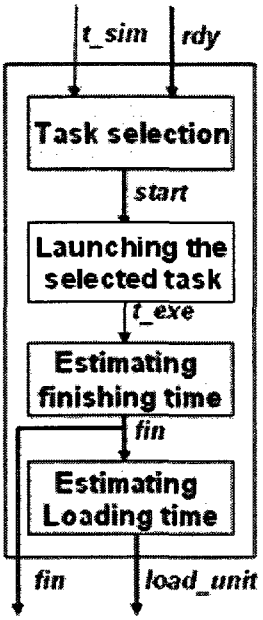

Figure 23 Block diagram of a Slave.

At the Task selection functionality level, the situation of all tasks placed on the Slave is verified to select one task to be executed. It is possible that no task is accepted to be executed at a given simulation stage. In this case, the other presented functionalities of the Slave are not performed. The Task selection step consists of selecting a task to be executed among all the admissible tasks. Here, an admissible task is a task belonging to the target Slave (*task_unit*(i) = *slave_nb* where *slave_nb* represents the Slave number) and that is ready for execution, meaning that all previous tasks (or

data) have been executed (or received), and that its ready time does not exceed the actual simulation reference time ($-1 < rdy(i) \leq t\_sim$). In each Slave unit, the corresponding assigned tasks with a ready time greater than *t_sim* are not considered for scheduling. These tasks are not ready yet and will be considered in the following simulation stages. The necessity of such limitation for accepting tasks is demonstrated by a detailed example later.

It is possible that several tasks are recognized as admissible ones at the Slave. Thus, one of these admissible tasks is selected for execution based on a scheduling algorithm, which is included in the corresponding scheduler on the Slave. Thus, the start time (*start*) of the selected task is determined, which is equal to the current simulation time (*t_sim*).

Emulating the launch of selected task

After selecting a task and determining its execution start time, the launch of the task is emulated and its execution time is estimated. Here, we use the term emulate to underline the fact that the task is not actually executed. Instead, we just estimate the time taken by the corresponding resource to execute this task. The estimated execution time of the selected task is presented by the parameter *t_exe*.

Estimating the completion time of task

Afterward, the completion time of the executed task is determined. Since the simulation time at the current simulation stage is equal to *t_sim*, the completion time of the task is estimated using Equation 5, where *index_run* is the executed task number.

$$fin\ (index\_run) = t\_sim + t\_exe \qquad (5)$$

Estimating the loading time of a Slave

The loading time of the Slave unit (the time from which a Slave is ready to execute another task) is equal to the completion time of the executed task as expressed in

Equation 6. In this equation,*slave_nb* and *index_nb* respectively denote the Slave number and the executed task number.

$$load\_unit\ (slave\_nb) = fin\ (index\_run) \qquad (6)$$

The pseudo code of a Slave is shown in Fig. 24, which represents all the functionalities performed by a Slave unit.

```
Pseudo code of a Slave unit:
Assigning the Slave number which is represented by the parameter slave_nb.

if loading time of Slave is equal or less than t_sim
   (Task selection)
   for i = 1 to m
      (Verifying if the tasks is assigned to the Slave, if the task is ready & if its ready time is
      equal or less than t_sim)
      if tsk_unit (i) = slave_nb & rdy(i) > -1 &  rdy(i)<= t_sim
         (Scheduling)
          Verifying some parameters of task based on the corresponding scheduling algorithm
          to select a task for execution.
      end if
    end for
    (After the above loop, one of the admissible tasks on the Slave is selected for execution.
    Index of selected task is presented by the parameter index_run.)

    (Determining the start time of selected task which will be equal to the simulation time)
    start (index_run) = t_sim
    Executing the selected task which results the execution time of task (t_exe).

    (Estimating the completion time of the executed task.)
    fin (index_run) = start(index_run) + t_exe

    (Deactivating the start time and the ready time of executed task.)
    start (index_run) = -1
    rdy (index_run) = -1

    (Loading time estimating of the Slave)
     load_unit (slave_nb) = fin(index_run)
end if
```

Figure 24 Pseudo code of a Slave unit.

3.3. One performance model example

In order to better clarify the described functionalities of the Master and Slave units in a performance model, we present an example case, which is shown in Fig. 25. As shown in this figure, the considered performance model consists of the Master and two Slaves where the Slaves include four tasks with the task dependency information described in the shown *Dep* vector. In addition, we have simulated this model and we present the values of some model parameters used during the first fourth stages of simulation in Fig. 26.

As explained before, at each simulation stage, the Master first evaluates the *rdy* vector, and then the vector *st* is estimated, which leads to defining the simulation time (*t_sim*). Thus, Slave1 and Slave2 evaluate the timing parameters of their corresponding tasks, in order. Each Slave estimates the elements of vectors *start* and *fin* (which correspond to its assigned tasks), and one element of vector *load_unit* corresponding to the Slave is evaluated. Such estimated model parameters can be observed in Fig. 26, which have been defined based on the explained order of simulation. In the presented example, task1 and task2 do not depend to any task and they are activated by two individual input signals with arriving periods of 0.02s. The first arriving time of the input frames is 0s then the first ready times of task1 and task2 are 0s, which are presented in the vector *rdy* in the first stage of simulation as shown in Fig. 26.

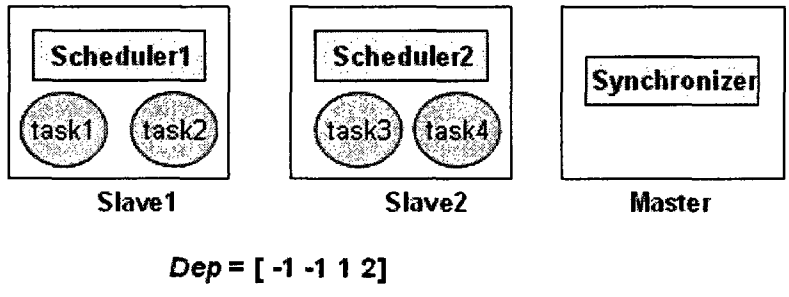

Figure 25 Block diagram of a performance model example.

		Initial case	Simluation stage 1	Simulation stage 2	Simulation stage 3	Simulation stage 4
Master	rdy	[-1 -1 -1 -1]	[0 0 -1 -1]	[0.02 0 0.01 -1]	[0.02 0.02 -1 0.02]	[0.04 0.02 0.03 -1]
	st	[-1 -1 -1 -1]	[0 0 -1 -1]	[0.02 0.01 0.01 -1]	[0.02 0.02 -1 0.02]	[0.04 0.03 0.03 -1]
	t_sim	0	0	0.01	0.02	0.03
Slaves	start	[-1 -1 -1 -1]	[0 -1 -1 -1]	[-1 0.01 0.01 -1]	[0.02 -1 -1 0.02]	[-1 0.03 0.03 -1]
	fin	[-1 -1 -1 -1]	[0.01 -1 -1 -1]	[-1 0.02 0.02 -1]	[0.03 -1 0.02 0.03]	[-1 0.04 0.04 0.03]
	load_unit	[0 0]	[0.01 0]	[0.02 0.02]	[0.03 0.03]	[0.04 0.04]
	Executed tasks	x	task1	task2 & task3	task1 & task4	task2 & task3

Figure 26 Parameter values in a performance model during four simulation stages

We now explain the estimations done by the Master and Slave units at the fourth simulation stage based on the results shown in Fig. 26. At stage 4, the Master first defines the ready time of the tasks. Based on Fig. 26, we observe that task1 has been executed at stage 3 then its ready time is updated at stage 4 by the next arrival time of the corresponding input signal, which is equal to 0.04s. Task2 has been ready from the previous stage and it has not been executed. Thus, the ready time of task2 has the same value as the one of stage 3, which is equal to 0.02s. Since task3 depends on task1 and task1 has been executed at stage 3, the ready time of task3 takes the previously estimated completion time of task1, which is equal to 0.03s. Task4 depends on task2 which has not been executed at stage 3. Thus, task4 is not ready for execution at stage 4. After estimating the elements of vector *rdy* by the Master, the earliest start time of tasks are evaluated considering their ready time and the loading time of corresponding Slave units. Thus, the simulation time of the current stage is evaluated which is the minimum value between the elements of vector *st* and is equal to 0.03s.

After the mentioned evaluation by the Master unit, Slave1 is simulated. As determined by the Master, task1 and task2 on the Slave1 are ready at stage 4. After the limitation process by Slave1, task1 (with a ready time greater than *t_sim*) is not considered as an admissible task and only task2 is accepted. Thus, task2 is selected and executed by Slave1. The start time of task2 is updated by the value of the simulation time, which is equal to 0.03s. Note that the execution time (*t_exe*) of all tasks in this

example is set at 0.01s. Therefore, the completion time of task2 takes the value of 0.04s. Consequently, the loading time of Slave1 is estimated to be equal to the completion time of task1, which is 0.04s. Consequently, Slave2 is simulated while it includes only one ready task (task3). Since the ready time of task3 is equal to the simulation time, this task is selected and is executed by Slave2, which provides the corresponding start and completion time. At the end, the loading time of Slave2 is evaluated.

We now describe the reason why the Slaves verify if the ready time of their ready tasks does not exceed the simulation time. As an example, in the presented performance model at stage 4, Slave1 includes two ready tasks with ready times of 0.04s and 0.02s. If Slave1 does not perform such verification, both tasks will be admitted, which leads to selecting one of them while the ready time of task1 exceeds the actual simulation time (*t_sim* – 0.03s) and it is not actually ready at the current simulation time. Therefore, not considering such verification can lead to selecting and execute task1 instead of task2, which provides the wrong task execution sequence. The simulation results for this example, including 25 simulation stages, are presented in appendix 1.

3.4. Conclusion

In this chapter, the basic concepts of the proposed performance modelling methodology were explained. We presented the three modelling steps needed to create a performance model, which allows us to abstract the application and architecture properties and to structure a high-level mixed description of the hardware and software. The performance model includes several units representing different architecture resources where each unit included several tasks representing different processing segments of the target application. In addition, the model units were structured in a master/Slave form, where the Master unit was responsible for managing the execution of tasks and the Slave units were responsible for executing the corresponding tasks.

Thus, we explained in details the functionalities of the Master and Slave units along with different timing parameters included in the performance model. At the end,

one example of performance model was demonstrated and the simulation results for this example were presented, which illustrated how to estimate the different timing parameters during simulation. In this way, we described the technical specification of our performance modelling method in details and showed how an actual performance model could provide the detailed timing information needed to allow executing an application on a target MPSoC platform without performing any functionalities of the application. In addition, we showed the manner in which such a model includes the mapping and scheduling strategies.

CHAPTER 4

SCHEDULING OF TURBO DECODING

In this chapter we discuss firstly the concept of mapping the uplink WCDMA processing corresponding to a UMTS base-station receiver on an MPSoC platform. We propose a mapping strategy to assign the different processing segments of the target application on the multiprocessor platform in such a way the WCDMA processing on the data blocks of each user is performed on different processors in a pipeline manner. In the proposed mapping method, it is supposed that each processor is dedicated to perform only one type of processing (such as Rake, Rate matching, Turbo decoding, etc) on the data blocks of several users. Since all the data blocks arrive to a UMTS base-station periodically, each uplink WCDMA processing which should be performed on the data blocks of a user can be considered as a periodic function where each processing segment of the function such as Turbo decoding is considered as a task. Also it is supposed that a synchronization protocol is included in the system that makes all the tasks be executed on the processors periodically.

Afterward we focus on the processors of the platform which are dedicated to perform the Turbo decoding process. Then processing variability of the Turbo decoding is discussed and the BER performance of such decoding is characterized. Thus we discuss the scheduling methods of Turbo decoding on the processors dedicated to this process. We propose some flexible scheduling methods which are adapted to the variable characteristics of the Turbo decoding. To investigate the proposed methods, we utilize our developed performance modelling methodology and we create a performance model including the developed flexible scheduling methods and the tasks representing the Turbo decoding processes. These tasks include the discussed characteristics of the Turbo decoding.

Simulating such a model provides the BER performance results for the decoded blocks in the different cases of proposed scheduling methods. Comparing the obtained results with a Worst Case Execution Time (WCET) design shows the advantage of the proposed flexible scheduling methods to improve the utilization of processors. Afterward we describe a method to validate the utilized manner of modelling the Turbo decoding process. Finally we present the elapsed simulation times in two cases of functional and performance modelling of Turbo decoding. The presented simulation times show the advantage of our performance modelling method which allowed rapid verification of the different scheduling methods without any functional simulation.

4.1. Mapping the uplink WCDMA processing on an MPSoC platform

It is mentioned that, in the UMTS receiver base-station, the radio frames arrive at a rate defining a processing period, where each radio frame is a concatenation of (transport) blocks, where each block is associated to a given user. As described in section 2.1, in a UMTS receiver base-station, the uplink WCDMA processing shown in Fig. 3, should be applied on the received data blocks. It is supposed that each data block corresponds to one individual user.

In order to simplify the problem, we suppose that the uplink WCDMA processing which should be applied on the data blocks of different users, is composed of three sequential processing segments. We present now an example of a UMTS receiver base-station which includes the uplink WCDMA processing on the data blocks of ten individual users as shown in Fig. 27. To implement such an application on a multi-processor platform, we need a mapping strategy to assign the different processing segments into the processors. As a first mapping strategy example, we suppose that the processing segments corresponding to each user are assigned to different processors. Thus, the chain segments of each user can be executed in a pipeline fashion on different processors. Also, we consider that each processor executes only one type of processing

(namely a segment) for multiple users. The proposed mapping strategy is same as the method presented in [29].

An example of such mapping strategy is illustrated in Fig. 28 which corresponds to the case where the application in Fig. 27 is implemented on four processors. Thus based on the explained mapping strategy, we suppose that each processor of the platform is dedicated to perform only one type of processing segment such as Rake, Rate matching, Turbo decoding and etc. on the received data blocks of different users.

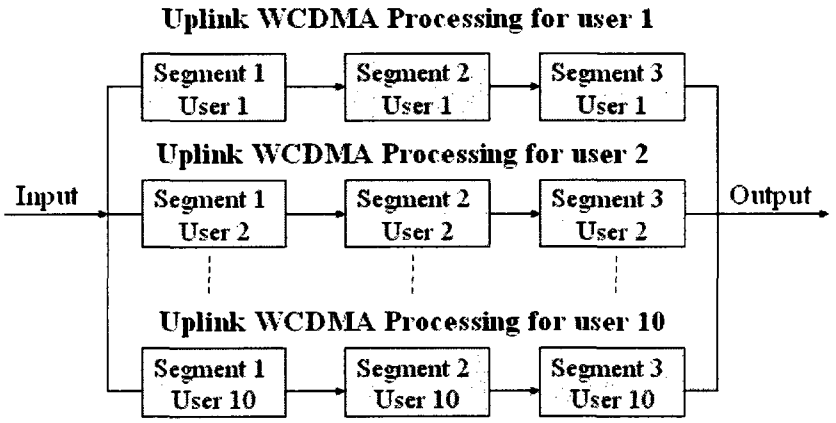

Figure 27 Example of processing on an UMTS receiver base-station.

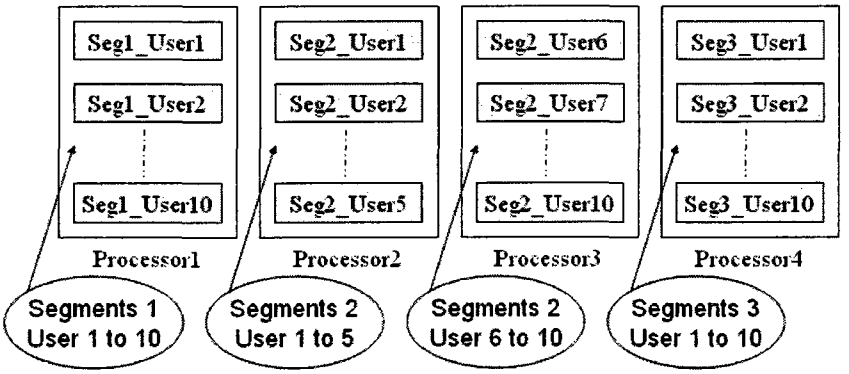

Figure 28 Mapping Example.

In section 2.5, a case study was presented to implement several periodic end-to-end functions on a multiprocessor platform where each function was composed of several sequential tasks. In that section, it was explained that after assigning the different tasks to the processors of the platform, we need a synchronization strategy to manage the execution of tasks on the processors in such a way that the precedence relations between the tasks were respected. Thus some synchronization strategies such as greedy and non-greedy protocols were introduced where the non-greedy protocol could make the tasks periodic. In our case study, each uplink WCDMA processing which should be performed on the data blocks of one user is considered as a periodic end-to-end function and consequently each processing segment are considered as a task. After assigning the tasks to different processors of our target platform, we suppose that a non-greedy synchronization protocol is utilized in the system which makes all the tasks periodic.

In the following subsections, we focus on the Turbo decoding process and the scheduling algorithms on the processors of the platform which are dedicated to perform this kind of process are studied.

4.2. Processing variability of the studied Turbo decoder

In this section, the processing variability in a Turbo decoder is discussed. The studied Turbo decoder in this project includes the specifications presented in section 2.1.4.1. As explained in that section, the processing variability of Turbo decoding comes from the varying number of decoding iterations required to complete the process. In order to analyze the methods for scheduling of Turbo decoding which will be presented in section 4.4, we need a good estimate of the variability detail of a decoding process. To investigate the processing variability of Turbo decoding, we utilized a complete Simulink model which has been developed by our team to accurately represent the whole Turbo coding/decoding process. In this model, the communication channel has been represented by an Additive White Gaussian Noise (AWGN) block, while the coder and the decoder have been implemented by several Matlab based functions. By

simulating this Simulink model, we extracted the effective required number of decoding iterations in a Turbo decoder to be performed on the signals received under different channel conditions.

Typical results for the number of iterations obtained under different channel conditions are presented in Fig. 29. The observed probability distributions are similar to Poisson distributions. Note that, because the number of iterations is hard limited between 2 and 8, the shape of the distribution is distorted (notably the tail). It is mentioned that hard limiting the number of iterations between 2 and 8 is forced by our studied Turbo decoder based on its structure standard [4]. The average number of iterations was also estimated in different channel conditions as reported in Fig. 30. This figure shows a decrease in the average number of decoding iterations as the channel condition improves, namely when Eb/N0 increases.

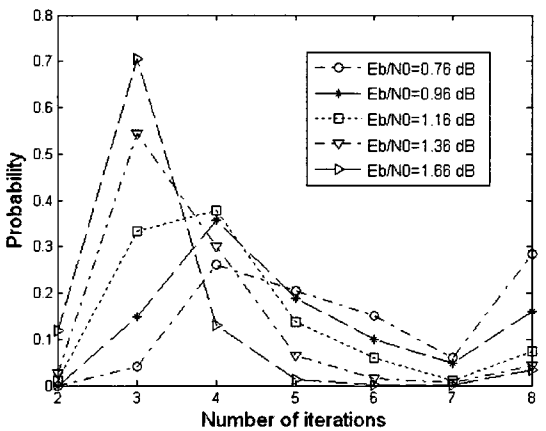

Figure 29 Probability density of the number of iterations.

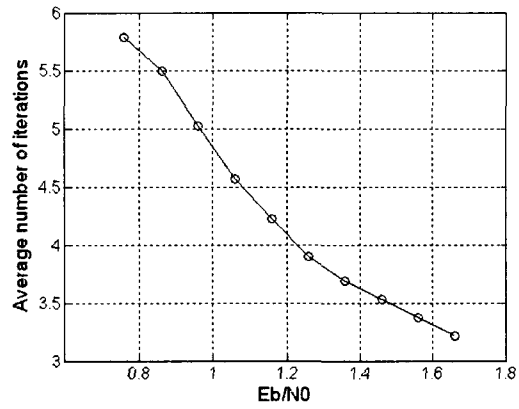

Figure 30 Average number of decoding iterations.

4.3. BER performance of the studied Turbo decoder

We characterize now the BER performance of our studied Turbo decoder in different channel conditions. As explained in the previous subsection, the number of decoding iterations performed by the decoder can vary but it is hard limited to the maximum value of 8 iterations. Let us call *it_eff* the number of iterations performed by the decoder in a normal condition which has the variability characteristics shown in the Fig.29 and Fig.30. As our scheduling strategies may force the decoder to perform fewer iterations that it would otherwise, we also need to characterize the impact of reducing the number of decoding iterations, in terms of additional errors affecting decoded blocks. Let us call *it_max_perm* the maximum number of decoding iterations permitted to the decoder, which can be set to a value between 3 and 8. The *it_max_perm* is a parameter introduced in the Turbo decoder that can limit the number of performed decoding iterations to a value lower than the effective number of iterations (*it_eff*).

We simulated the previously validated Simulink model of Turbo coding/decoding process with a sufficient number of frames such that the total number of processed bits is set to about 100 times the inverse of the target bit error rate in our simulations for all considered sets of operating conditions. The simulations were repeated for each considered *it_max_perm* to different value from 3 to 8. During simulations we estimated

the number of bits in error in the decoded blocks. These simulation results are listed in Table 3. This table includes the Bit Error rate (BER) results when the effective decoding are not limited (*it_max_perm* = 8). Also it includes the average number of additional bits in error (*ave_add_err*) in the decoded blocks which are caused by limiting the effective decoding in different cases of *it_max_perm* from 3 to 7 comparing to the case when the *it_max_perm* is set to 8. It is mentioned that the unit of *ave_bit_err* values shown in Table 3 is bit.

Table 3 Performance parameters of Turbo decoder.

Eb/N0	Number of Frames	BER (it_max_perm=8)	ave_add_err (it_max_perm=7)	ave_add_err (it_max_perm=6)	ave_add_err (it_max_perm=5)	ave_add_err (it_max_perm=4)	ave_add_err (it_max_perm=7)
0.76	7	2.10 e-2	1.59	2.53	3.66	8.70	14.14
0.86	13	1.16 e-2	4.07	7.86	9.11	9.33	13.11
0.96	25	5.90 e-3	5.72	9.19	10.51	9.25	11.03
1.06	52	2.90 e-3	1.85	2.78	6.04	8.99	9.07
1.16	123	1.23 e-3	-0.45	2.60	4.58	3.94	5.05
1.26	337	4.52 e-4	1.43	1.34	2.40	3.67	3.42
1.36	1029	1.48 e-4	0.50	0.34	1.08	1.98	2.10
1.46	3623	4.20 e-5	0.09	0.56	0.96	1.43	1.43
1.56	15151	1.00 e-5	0.17	0.27	0.46	0.85	0.87
1.66	99633	1.53 e-6	0.03	0.09	0.19	0.37	0.47

4.4. Proposed methods for scheduling the Turbo decoding

In this section we focus on the platform processors which are dedicated to perform the Turbo decoding process on the received blocks and we discuss the scheduling concepts on these processors. In order to simplify the problem, we consider the scheduling of Turbo decoding on only one processor. We assume that all received frames in the considered UMTS receiver base-station have the same arriving rate defining a processing period. To determine the number of data blocks (users) that can be assigned to the processor during each processing period, we need to estimate the execution time of corresponding Turbo decoding processes on the target processor. We suppose that total execution time of all assigned Turbo decoding processes on each processor should not exceed the processing period. In this way all the decoding processes will be finished by the processor before arriving data blocks of the next period.

Considering the significant processing variability of Turbo decoding, we expect that the worst case estimating of execution time would be highly inefficient. In order to improve the resource utilization, we consider that the allocated time to each Turbo decoding process is smaller than WCET case and consequently we can assign more number of data blocks (users) to the processor. However, it is possible that, in some cases, processes require more than their nominal allocated slots due to channel degradation which can lead to scheduling problems. To resolve this issue, we propose some flexible methods of scheduling which are associated with control strategies that can limit the effective processing to meet the timing constraints. Descriptions of the proposed methods are presented in the following.

To investigate the proposed scheduling methods, we utilize our performance modelling method explained in chapter two. As mentioned earlier, we intend to focus on scheduling of Turbo decoding processes on one processor of the platform. Thus we create a performance model including one Master and one Slave unit where the Slave represents one processor of platform which is dedicated to Turbo decoding processes and the Master including the Synchronizer module corresponds to another processor of platform. Based on the number of data blocks (users) assigned to the processor, several tasks representing the Turbo decoding processes are included in the Slave. The Turbo decoding processes are modelled in the form of different tasks. Performance modelling of Turbo decoding is not based on detailed decoding, but rather on the previously reported service distributions (Fig. 29) that were obtained by detailed decoding. Each task representing Turbo decoding contains a random number generator which models the mentioned distributions. Simulating each Turbo decoding task (task representing Turbo decoding), provides the effective number of decoding iterations (*it_eff*) and consequently the execution time of corresponding decoding process on the target processor. It is mentioned that we have extracted before, the execution time of each decoding iteration using an Instruction Set Simulator of the target processor and introduced it in the performance model. Execution time of each decoding process is provided by the

performance model using execution time of each decoding iteration and effective number of iterations.

It is mentioned that when a data block is received to the processor to be decoded, its corresponding task on the Slave is supposed to be activated. In other words, the task becomes ready. Thus all tasks assigned to the Slave have the same activation period which is equal to the processing period. We suppose that all tasks become activated or ready at the same time in all processing periods.

4.4.1. One shot scheduling

In the first proposed scheduling method, we suppose that a number of Turbo decoding tasks are assigned to the Slave. Then, based on the number of tasks assigned to the Slave and the processing period, the dedicated resources to perform the decoding emulated by each task are determined. From that point, we make the simplifying assumption that channel conditions change slowly and that all blocks of the received frames have the same Eb/N0, thus all tasks are given the same resources. Considering the allocated resource for each Turbo decoding task, a guaranteed budget of decoding iterations available to emulate the decoding of each block is determined by the system which is called Iteration Budget (*IB*). As explained next, this budget may change during a processing period for each task and is limited between 3 and 8.

During a processing period, the scheduler placed on the Slave selects all the ready tasks of the Slave in a round robin manner to be simulated. After simulating each Turbo decoding task, if the provided effective number of decoding iterations (*it_eff*) is greater than its *IB*, the number of emulated decoding iterations for the corresponding block is considered to be equal to *IB*. The corresponding block to which the scheduler has not allocated its *it_eff* before the end of a processing period is considered to be partly decoded. Due to the iterative nature of Turbo decoding, a partly decoded block probably contains some additional errors. The number of additional errors is estimated by using Table 3 which has been introduced in the performance model. If the allocated budget is

equal or greater than *it_eff*, the number of emulated iterations is considered equal to the corresponding *it_eff*. A block that can reach its *it_eff* is fully decoded.

Indeed if *it_eff* is less than the budget for a task, the difference is distributed between unsimulated tasks in the current processing period and is added to their previous allocated budgets. This redistribution is done as uniformly as possible. Fig. 31 shows the one shot scheduling manner on a Slave including four Turbo decoding tasks during one processing period. The task *IB* values are updated before simulating each task as shown in Fig. 31. In this example, in each processing period, there is a total of 17 decoding iterations as a budget to decode four corresponding data blocks in each period. Before simulating the first task, the total number of iterations (17) is distributed over the tasks as shown in the second column in Fig. 31. Simulating each task provides the corresponding *it_eff*. Afterwards the *IB* value of the tasks not yet simulated is updated. As shown in Fig.31, data blocks of tasks 3 and 4 are effectively processed while the corresponding decoding of 2 other tasks are degraded in order to respect the assigned resource budgets.

Pseudo code of one shot scheduling method for a general case of *n* Turbo decoding tasks on one Slave is presented in Fig. 32. Variable *grade* shown in this figure is considered to distribute not used iteration budgets between the tasks which have not been simulated.

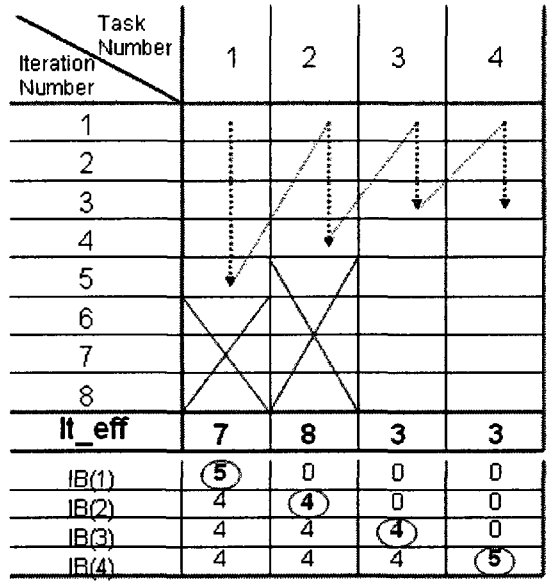

Figure 31 One shot scheduling example.

```
Initializing IB of all tasks
grade =0
Task selection (The ready tasks are selected for simulation in a
                round robin manner. We suppose that task i is selected)

task i is simulated and it_eff (i) is obtained.
if (it_eff (i) > IB(i))
    The emulated number of iterations is limited to IB(i). Thus the
    emulated decoding by task i is degraded from the effective one
    and corresponding additional errors are estimated by using table I.
else
    grade= IB (i) - it_eff (i)
end if
while ( grade ~= 0)
    for j=i+1 to n
        if (grade>0)
            IB (j) = IB (j) +1
            grade= grade -1
        end if
    end for
end while
```

Figure 32 Pseudo code of one shot scheduling.

4.4.2. Gradual scheduling

As explained in one shot scheduling method, during each processing period, the unused portion of the iteration budget of each simulated Turbo decoding task is distributed and added to the previous allocated budgets of not yet simulated tasks. In this way the tasks which are simulated later are advantaged with respect to the tasks simulated earlier as their *IB* might be increased. Because of random quality of corresponding blocks, it is possible that the later simulated tasks do not use totally their increased *IB*s while the earlier simulated tasks need more iterations. The particular scenario occurs in Fig. 31 example, where tasks 1 and 2 are incomplete and would requires additional iterations while task 4 does not utilize totally its *IB* which is increased by 1.

In order to optimize the resource allocation of tasks, another method called gradual scheduling is proposed. In this method, a global *IB* is allocated to all tasks which is equal to the number of decoding iterations that could be performed by the target processor during one processing period. In the gradual scheduling method, the scheduler selects the ready tasks for simulation in a round robin manner. After emulating one decoding iteration for each selected task, the task's simulation is preempted and consequently the next ready task is selected to be simulated in the same manner.

Fig. 33 illustrates the gradual scheduling method on a Slave including four Turbo decoding tasks during one processing period. As shown in Fig. 33, the initial value of global *IB* is equal to 17 which is equivalent to sum of initial *IB*s of tasks in the one shot scheduling example. After emulating one decoding iteration of each task, the global *IB* is decremented by one. A task is no longer simulated when it reaches its effective number of decoding iterations. Simulation of the remaining ready tasks continues in the same way until the global budget becomes zero. After termination of global IB, simulation of ready tasks which have not reached to their effective processing are stopped which leads to the degradation of processing on the corresponding blocks. In Fig. 33, the effective

decoding for task 1 and task 2 is degraded, similarly to the one shot example. But now, one additional iteration is granted for the mentioned tasks.

In the gradual scheduling method, the number of emulated iterations for the ready tasks at each simulation moment is approximately the same. Thus the corresponding blocks of the stopped tasks which have not reached to their effective number of iterations, have almost the same processing level and consequently their processing are degraded approximately identically. In other words the resource allocating to the tasks with degraded processing is nearly uniformed. Another advantage of this method comparing to the one shot method is that the global *IB* is utilized completely in the case of processing needs for the ready tasks.

Pseudo code of gradual scheduling method for a general case of n Turbo decoding tasks on one Slave is presented also in Fig. 34.

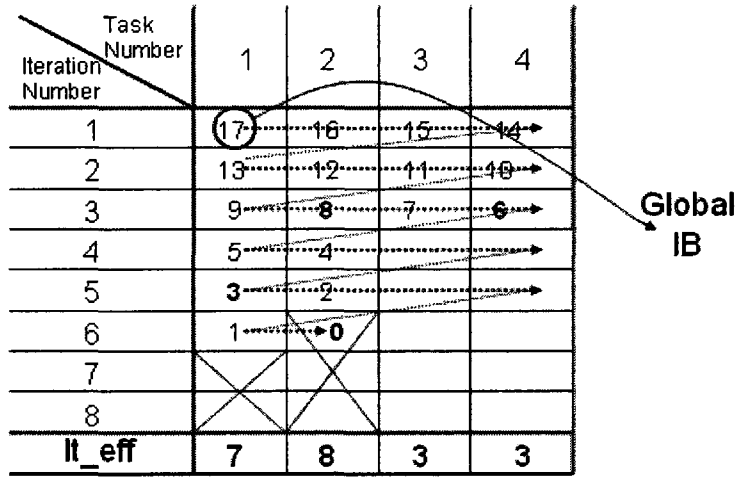

Figure 33 Gradual scheduling example.

```
Initializing the globalIB

Tasks selection. (In this method, instead of selecting one task,
                  all ready tasks on the slave are selected.)

while( globalIB ~= 0)
    for i=1 to n
        if task i has been selected & globalIB ~= 0
            1- One decoding iteration of task i is emulated.
            2- globalIB is decremented by 1.
            3- Simulation of task i is terminated if the task has
               reached to its effective decoding level.
        end if
    end for
end while

for i=1 to n
    if  Simulation of task i has not been terminated
       task i is terminated and its residual errors caused by the
       degradation in the emulated decoding, are estimated by using
       table I.
    end if
end for
```

Figure 34 Pseudo code of gradual scheduling.

4.4.3. Priority-driven one shot scheduling

As explained before, the earlier simulated tasks might be allocated smaller IBs causing more additional errors, because of the degradation in the emulated processing compared to the later simulated tasks. Since the ready tasks are selected and simulated in a fixed order in all processing periods, the additional errors associated to the earlier simulated tasks are increased over time much more than for the tasks which are simulated later.

In this way the difference between the provided service qualities for the corresponding blocks (users) is increased over time. In order to prevent the increase of difference between the user's service qualities, we propose a modification in the one shot scheduling method. In the modified method, which is called priority-driven one shot scheduling, we assign a priority to the ready tasks to determine their order of

simulation during each processing period. In the model, we have considered a parameter called *sum_add_err* which is assigned to each task and represents the accumulated additional bits in error inserted into the corresponding decoded block from beginning of simulation till current processing period. It is mentioned that the additional bits in error represent the bits in error provided because of degradation in the effective decoding process, which occurs when the resulting it_eff of a task is greater than the corresponding IB and that the IB is less than 8, In this case the additional bits in error inserted into the corresponding decoded block are estimated to be equal to the corresponding *ave_add_err* extracted from Table 3. Then the corresponding parameter *sum_add_err* for the task is updated. The estimated parameter *sum_add_err* for each task determines its priority for simulation in the next processing period.

In the priority-driven one shot method, the method of updating the IBs of tasks is same as the one shot scheduling as shown in Fig. 31. But in this method, the order of placing the tasks on the columns shown in Fig. 31 is defined based on the priority of tasks. At each processing period, the ready tasks with the lowest values of *sum_add_err* have the highest priority and are simulated first. In this way the tasks which are characterized with higher accumulated bits in error are likely to be allocated more resource in the current processing period. Thus the provided quality of service for different users becomes more uniform when compared to the one shot scheduling method.

4.4.4. Priority-driven gradual scheduling

In the gradual scheduling, the tasks simulated later are on average allocated less resource than the ones simulated earlier. Thus more additional errors are associated to the later decoded blocks because of the degradation in the emulated processing compared to the earlier decoded blocks. For example, as shown in Fig. 33, task 1 which is simulated earlier is allowed 6 decoding iterations while the number of emulated iterations for task 2 is 5. Since the ready tasks are simulated in a fixed order in all

iteration levels and all processing periods, the number of additional errors associated to the later simulated tasks in different iteration levels is higher than the one associated with the tasks which are simulated earlier. In this way the difference between the provided service qualities for the corresponding users increases over time.

In order to prevent the increase of difference between the user's service qualities, we propose a modification in the gradual scheduling method. In the new modified scheduling method, which is called priority-driven gradual scheduling, we introduce the priority based on the *sum_add_err* parameter (similarly to the priority-driven one shot method): the parameter *ave_add_err* and consequently the parameter *sum_add_err*, is estimated and associated to each totally simulated task in each processing period which defines the priority of the task in the next period. Otherwise, as for the gradual scheduling, the tasks are simulated gradually and a global IB is assigned to limit the emulated processing of tasks as shown in Fig. 33.

In other words, in this method the order of placing the tasks on the columns shown in Fig. 33 is based on the priority of tasks. At each processing period, the ready tasks with the highest *sum_add_err* value have the highest priority for simulation and are simulated first. In this way the tasks with more accumulated additional bits in error should be allocated more resources in the current processing period. Thus the provided quality of service for different users becomes more uniform compared to the gradual scheduling method.

4.5. Simulation results

As explained earlier, we create a performance model to investigate the proposed scheduling methods which includes one Master and one Slave unit. Thus we develop the previously proposed scheduling methods and introduce them in the Slave unit as different options to schedule the assigned Turbo decoding tasks. Afterward, we simulate the model in different cases of proposed scheduling methods under various loads and channel conditions for appropriate durations. The simulation of the model provides the

BER associated to the decoded blocks in all processing periods. In order to simplify the results, we calculate the average BER of all corresponding tasks during the simulation in different cases of system loads and channel conditions. The specific Turbo decoding tasks correspond to 656 data bits blocks before encoding ((3*656) +12= 1980 symbols after encoding)), and frames are transmitted at 64 ksymbol/s over 40 ms periods.

If we apply the WCET design, considering the period of received frames and the worst case processing time of Turbo decoding on the target processor which includes 8 decoding iterations, we can assign only 14 data blocks (users) to the processor. Therefore, in the created performance model, if we assign 14 tasks to the Slave, 8 iteration budgets can be assigned to each task and no processing degradation is forced to the emulated Turbo decoding processes by the system. Thus, case of assigning 14 users (WCET design) is used as a reference to be compared with the case of assigning more users in different cases of proposed scheduling methods.

4.5.1. One shot scheduling

By simulating the mentioned performance model in the case of one shot scheduling method, the average BER of all tasks (corresponding to different users) are obtained in different cases of number of assigned users and channel conditions which are shown in Fig. 35. As shown in this figure, assigning more than 14 users causes an increase in the error rate. The horizontal distance between the average BER curves and the reference one (case of 14 users) for a given BER, gives the average degradation of the decoding gain. Based on Fig. 35, the average degradation of decoding gain is obtained for 3 different numbers of users at the BER value of 2*10-5. These results are reported in Table 4. Clearly, the average degradation of decoding gain increases with the number of users. However, even with 29 users assigned to a processor, the decoding gain degradation is only approximately 0.15 dB, which is negligible and does not have significant effects on the quality of service. When compared to the WCET strategy, which lets a system architect assign only 14 users per processor in the modelled conditions, the proposed one shot scheduling method allows assigning twice as many

users while providing acceptable quality of service. This advantage of the proposed scheduling method is obtained by effectively exploiting the significant variability of Turbo decoding process.

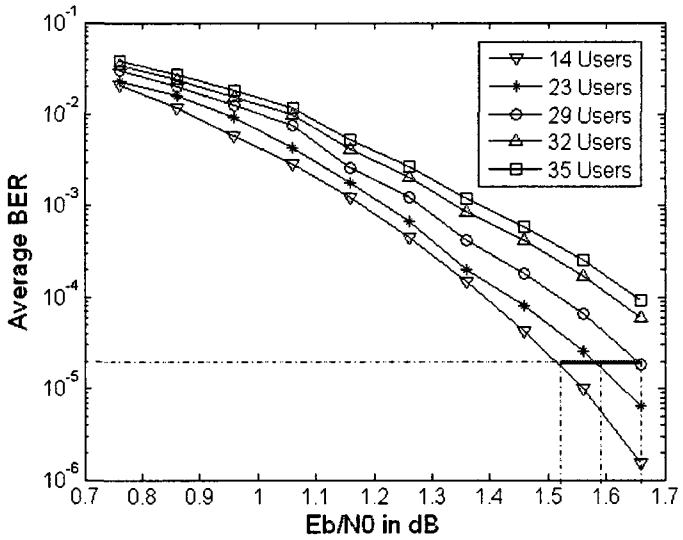

Figure 35 Average BER in case of one shot scheduling.

Table 4 Average decoding gain degradation at a BER of $2*10^{-5}$ for one shot scheduling.

Number of users	14	23	29
Average degradation of decoding gain (dB)	0	0.07	0.15

4.5.2. Gradual scheduling

By simulating the mentioned performance model in the case of gradual scheduling algorithm, the average BER of users is obtained in different cases of number of assigned users and channel conditions which are shown in Fig. 36. Based on Fig. 35 and Fig. 36, the gradual scheduling method provides a remarkable improvement of average BER over the one shot scheduling method. By using Fig. 36, average degradation of decoder gain for 5 different numbers of users (at the BER value of 2e-5) is obtained and gathered on Table 5. It is noteworthy that the average degradation of decoder gains with 23 and 29 users is now zero while it respectively reaches 0.07 and 0.15dB for the same number of users with the one shot scheduling method.

Also based on table 5, the average degradation of decoder gain for more users such as 32 and 35 are negligible (≤0.10dB). As a comparison, average degradation of decoding gain for 35 users with gradual scheduling is by 0.05 dB lower than for 29 users with one shot scheduling. This improvement of decoding performance shows the efficiency of gradual scheduling method in resource allocation to the tasks compared to the one shot scheduling.

Table 5 Average decoding gain degradation at a BER of $2*10^{-5}$ in case of gradual scheduling.

Number of users	14	23	29	32	35
Average degradation of decoding gain (dB)	0	0	0	0.03	0.1

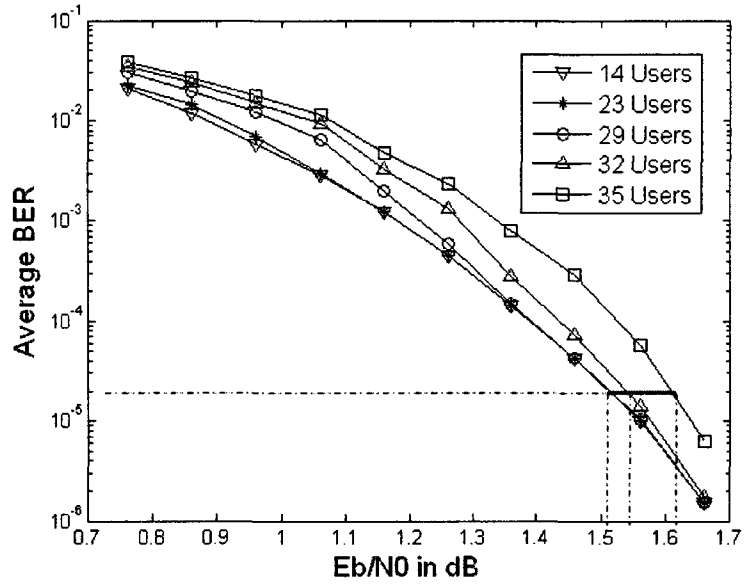

Figure 36 Average BER in case of gradual scheduling.

4.5.3. Priority-driven one shot and priority-driven gradual scheduling

By selecting the priority-driven one shot scheduling and then the priority-driven gradual scheduling in the mentioned performance model and simulating the corresponding models, the average BER of users in different cases of number of assigned users and channel conditions is obtained. By observing the results, we realized that the average BER results obtained by using priority-driven one shot method are the same as the corresponding results in the case of one shot method shown in Fig. 35. Also the priority-driven gradual method provides the same average BER results as provided by the gradual method and shown in Fig. 36.

In order to show the advantage of such priority-driven methods in equalizing the service qualities of users, we also estimated the variance BER of users. The BER variance of users in different cases of the proposed scheduling methods is presented in Fig. 37, Fig. 38, Fig. 39 and Fig. 40. These shown curves are parameterized by the number of users. In Fig. 39 and Fig. 40, the BER variance values for the Eb/N0 values which are greater than the values of the dotted lines are approximately zero. Looking at

Fig. 37 and Fig. 38, gives a first opportunity to compare the priority-driven and the plain one shot scheduling. As one can see, the priority-driven approach uniforms the BER performance of users by reducing BER variances with respect to the plain one shot method. The same observation can be made for the priority-driven and the plain gradual methods by comparing Fig. 39 and Fig. 40.

Comparisons are even easier to make by looking at Fig. 41, Fig.42 and Fig. 43, where the BER variance curves are drawn respectively for 29, 32 and 35 assigned users, parameterized by the type of scheduling method. In this way, in each case of mentioned number of assigned users, we can clearly observe the difference between the provided service uniformity for users in 4 cases of scheduling method.

Considering the shown results for the average BER and the BER variance, we find out the priority-driven gradual scheduling is the most efficient method between the proposed ones which provides the best BER performance and service uniformity for different users.

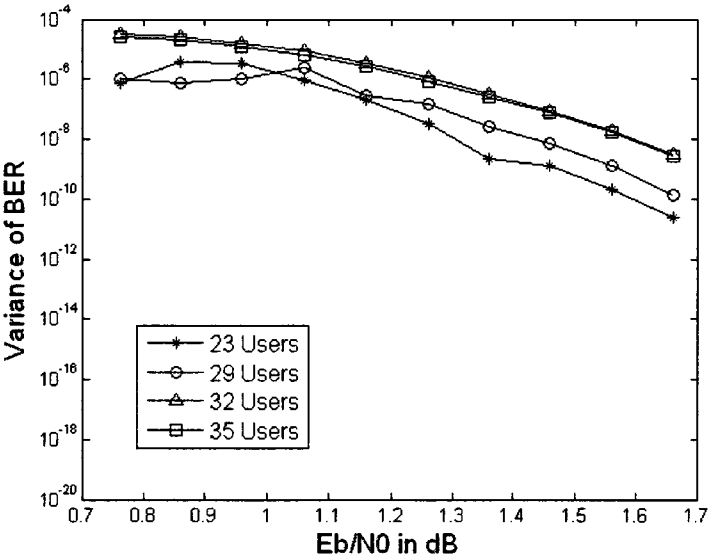

Figure 37 BER variance in one shot scheduling.

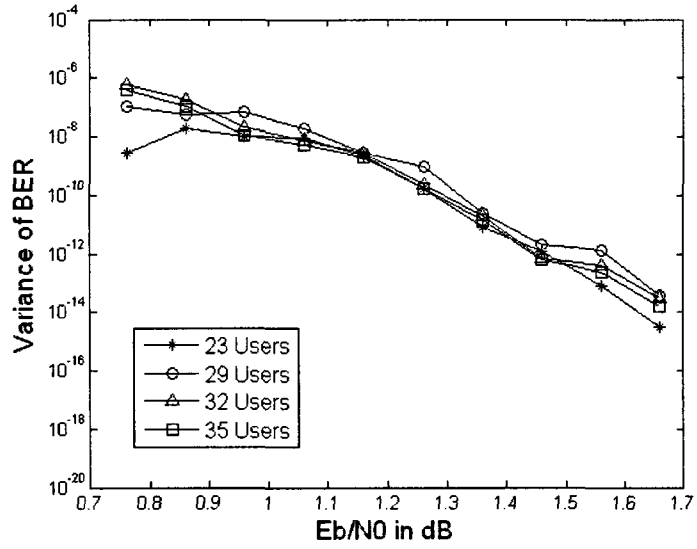

Figure 38 BER variance in priority-driven one shot scheduling.

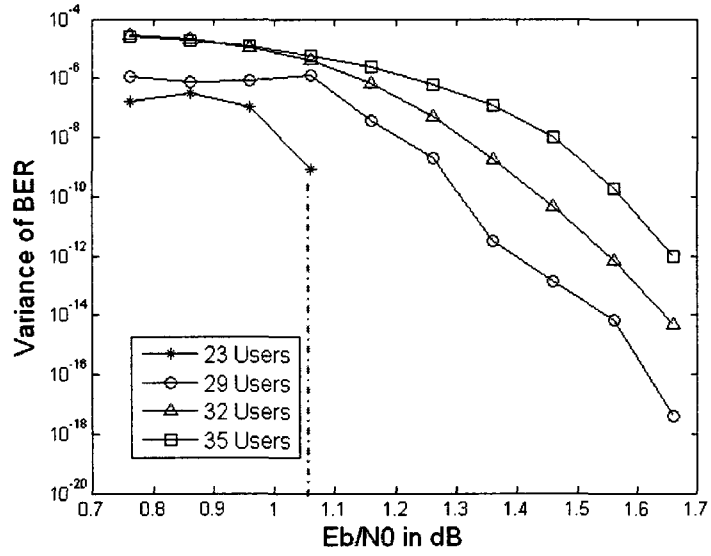

Figure 39 BER variance in gradual scheduling.

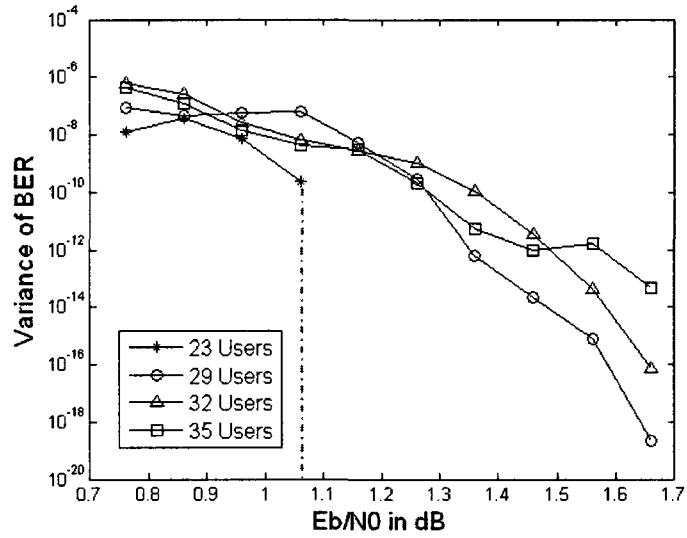

Figure 40 BER variance in priority-driven gradual scheduling.

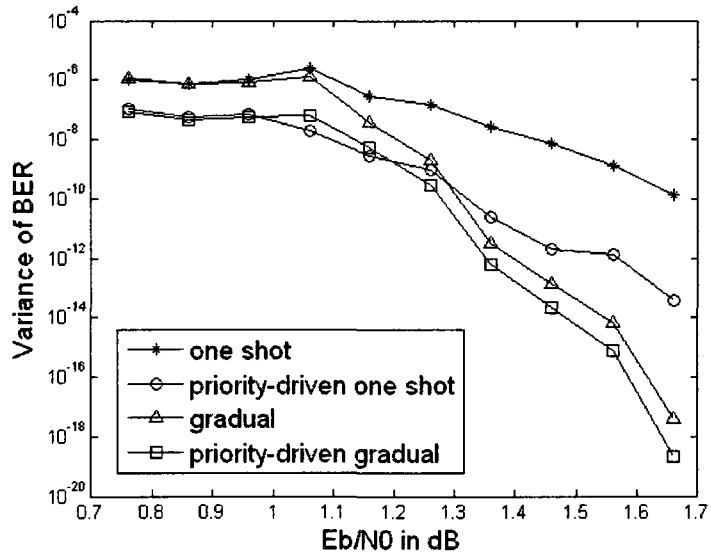

Figure 41 BER variance in case of 29 users.

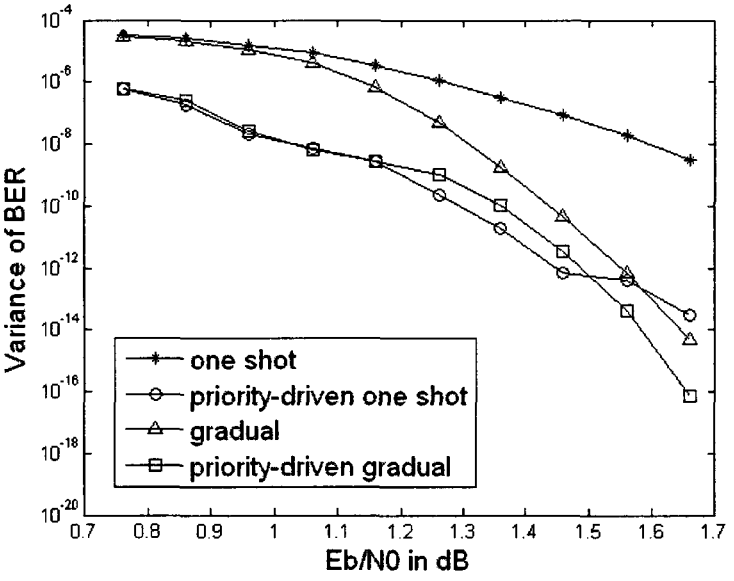

Figure 42 BER variance in case of 32users.

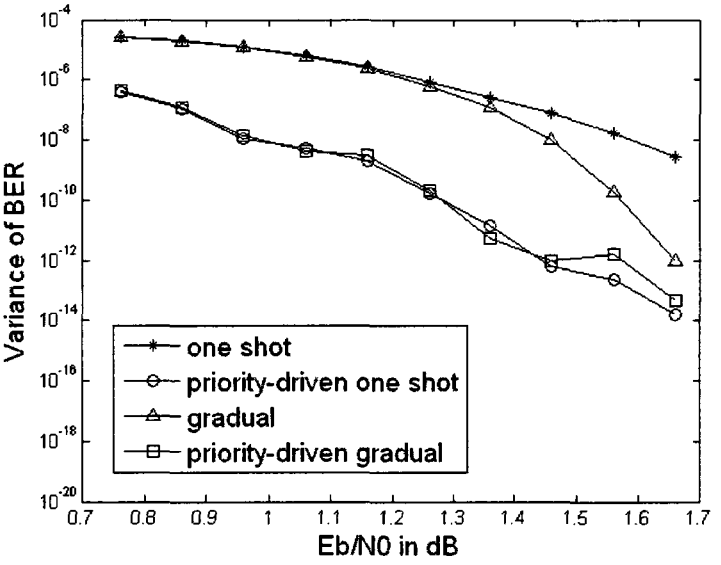

Figure 43 BER variance in case of 35 users.

4.6. Validating investigation

In this section we intend to validate the method that we utilized to model the Turbo decoding process and to estimate the residual errors in the decoded blocks. As explained in section 4.4, we designed a random generator to emulate the effective number of

decoding iterations required in a Turbo decoder, depending on the channel condition. We included such random generator in each task representing Turbo decoding in the performance model. Also in order to estimate the residual errors in the decoded blocks, we extracted some parameters such as BER and *ave_add_err* as illustrated in Table 3. During simulating the performance model, the effective number of decoding in the tasks representing Turbo decoding is provided by the corresponding random generators and the number of bits in error in the decoded blocks is estimated using Table 3.

In order to validate the method of modelling the Turbo decoding process, we developed a validation strategy. For this reason, we created several decoding tables by using the results of simulating the Simulink model (functional model) of Turbo coding/decoding process in different channel conditions. Such decoding tables include the effective number of decoding iterations and also the number of bits in error at all level of iterations till the effective one on different received data blocks for one user. Each decoding table corresponds to one channel condition and includes the mentioned decoding information for several data blocks. The number of blocks/frames included at each decoding table is selected in such a way that it is appropriate for the corresponding channel condition.

Thus, we do some modification in our performance model in such a way that we introduce the mentioned decoding tables in the performance model. In this modified version of the performance model, each task representing Turbo decoding process utilizes the decoding information included in the corresponding decoding table instead of using the random generator and the estimated number of bits in error in Table 3. Since at each channel condition, the received data blocks corresponding to different users can be different, we cannot use the same table for all tasks. Also, since providing the decoding tables in some channel conditions is time consuming, creating different decoding tables for individual users (up to 35 users in our case study), is not practical. Therefore, we did an approximation in our validation method in such a way that at each channel condition, each task uses the decoding information corresponding to a different

random combination of data blocks in the corresponding decoding table. In this way, at each channel condition, different tasks corresponding to different users utilizes the same decoding table, but at each time they take the decoding information corresponding to different data blocks of the table.

In this way, we develop the explained validation version of performance model including one Master and one Slave unit where the Slave contains several tasks corresponding to Turbo decoding process on the received data blocks of several users. Similarly to the explained simulations in section 4.5, we simulate this modified version of performance model in different cases of channel conditions and number of assigned users. We perform the explained simulations in two cases that the scheduling method in the Slave unit is set to the one shot scheduling and also gradual scheduling method. Therefore we obtain the average Bit Error Rate (BER) of different users as shown in figures 44 and 45 which correspond respectively to the cases of one shot scheduling and gradual scheduling methods.

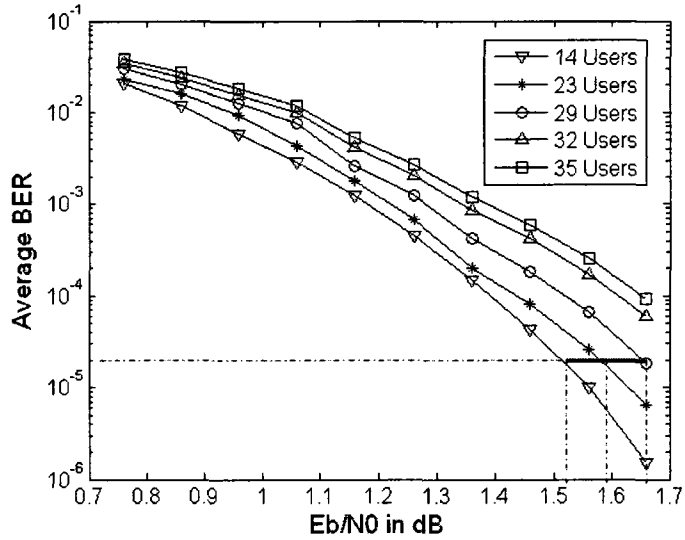

Figure 44 Average BER in case of validation model and one shot scheduling.

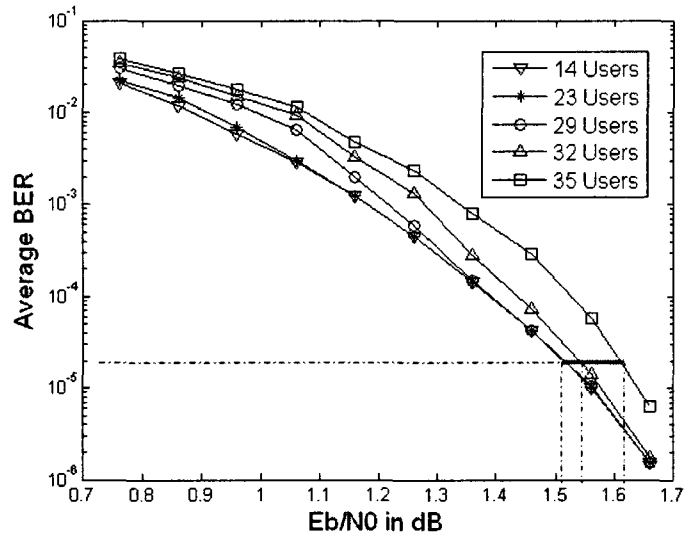

Figure 45 Average BER in case of validation model and gradual scheduling.

Comparing Fig.44 with Fig. 35 and Fig. 45 with Fig. 36 does not show any difference between the average BER results in two cases of using performance model and validation model. Indeed, there is a little difference between the values of average BER in these two cases of modelling, but the difference is such little that it is not obvious by comparing the results in the mentioned figures. In this way the utilized method of performance modelling of the Turbo decoding process is validated.

4.7. Elapsed simulation time

In this section, we intend to compare the elapsed simulation times in two cases of performance modelling and functional modelling of the Turbo decoding processes. Table 6 illustrates the simulation times in these two cases of modelling in different cases of channel conditions. As explained before, in different channel conditions, each Turbo decoding process should be simulated for a sufficient number of input data blocks/frames. For instance, the total number of processed bits is set to 100 times the inverse of the target bit error rate in our simulations for all considered sets of operating conditions. The numbers of blocks/frames in different cases of channel conditions are illustrated in the second column of Table 6.

Table 6 Elapsed simulation times.

Eb/N0	Number of Frames	Decoding Average Elapsed Time for one user (sec)	Decoding Average Elapsed Time for 35 users using functional model (sec) (T_sim_func)	Elapsed Time for scheduling 35 users using performance model (sec) (T_sim_perf)	T_sim_func / T_sim_perf
0.76	7	3.56	124.50	0.37	332.88
0.86	13	5.97	209.05	0.69	301.23
0.96	25	11.47	401.58	1.35	297.51
1.06	52	21.49	752.08	2.77	271.87
1.16	123	50.27	1759.36	6.60	266.76
1.26	337	141.87	4965.37	17.95	276.64
1.36	1029	470.31	16460.98	55.88	294.57
1.46	3623	1488.95	52113.37	195.05	267.18
1.56	15151	6604.64	231162.38	791.23	292.16
1.66	99633	42421.87	1484765.41	5346.45	277.71

Using the Simulink functional model of Turbo coding/decoding, we estimate the average simulation time for decoding the data blocks of one user for the appropriate number of blocks corresponding to the channel condition. These average simulation times are presented in the third column of Table 6. Thus we estimate the average simulation times for decoding the data blocks of 35 users in different channel conditions which are calculated by multiplying the corresponding simulation times for one user by the value 35 and are shown in the fourth column of the Table. Afterward, we consider the created performance model in section 4.5 which includes the one shot scheduling method and 35 tasks representing the Turbo decoding processes. Then we simulate the mentioned performance model in different cases of channel condition for appropriate number of blocks/frames and obtain the corresponding elapsed simulation times as shown in the fifth column of Table 6.

The Table 6 illustrates also the ratio of the simulation times in two cases of functional modelling and performance modelling for decoding the data blocks of 35 users. We can observe that the simulation times in the case of functional modelling are approximately 300 times of the simulation times in case of performance modelling. Such ratio of simulation times justifies the utilization of our performance modelling that allowed the rapid verification of different methods of scheduling without any functional simulation.

4.8. Conclusion

In this chapter, we discussed a mapping strategy to assign the different processing segments of the uplink WCMDA processing in a UMTS receiver base-station to different processors of a multiprocessor platform. Then we focused on the Turbo decoding process which was a high intensive computational part of the uplink WCDMA processing and we discussed the scheduling of Turbo decoding on the processors of the platform dedicated to this process. Thus, we used our developed performance modelling strategy to estimate the efficiency of the proposed scheduling methods. We created a performance model including the proposed scheduling methods and the tasks representing the Turbo decoding processes on the different data blocks.

Simulating such model provided the BER performance for the decoded blocks in different cases of scheduling methods and for different number of assigned users. The provided simulation results showed the advantage of these flexible scheduling methods comparing to a WCET design by increasing the number of assigned users in the base-station. Such flexible methods allowed the system to perform more number of processes by degrading gracefully the processing efforts which reflected the negligible service quality degradations.

In this chapter, similarly to [6] and [18], scheduling of Turbo decoding process on several encoded blocks in the dedicated processors was studied. However, our scheduling methods allowed much more flexible degradation by considering dynamic iteration budgets, when compared to the decoding degradation presented in [18]. Such processing degradation concept was not considered in [6]. Unlike to [8], [19] and [20], we considered that the Turbo decoder algorithm consisted of only one monolithic task and the Turbo decoding process on each coded block should performed totally on one processor. Thus, the Turbo decoding processes on the individual processors could be data independent which would reduce the data communication between the processors.

Finally, we provided the elapsed simulation times in two cases of functional and performance modelling of the Turbo decoding. These results showed the advantage of

using such performance modelling strategy to rapid verification of the proposed scheduling methods.

CHAPTER 5

CONCLUSION

In this project we utilized a multi-DSP platform as the target architecture which is called Vocallo and is fabricated by Octasic Semiconductor. We chose such a multi-core DSP platform to provide an appropriate target for implementing our telecommunication target application which had a dynamic and computationally intensive nature. The target application in this project corresponded to the WCDMA (Wide-band Code Division Multiple Access) process on the received data blocks corresponding to different users in a Universal Mobile Telecommunication Systems UMTS base-station receiver. Considering real-time and dynamic characteristics of target application, we derived high-level performance models of the application, using the same system-modeling environment (in our case Matlab/Simulink) to allow fast performance validation of that application when running on the target platform. Our devised performance modeling methodology also allowed validation of the efficiency of strategies for mapping and scheduling a complex application on the target platform before run time.

In this project, we focused specifically on a computationally intensive part of the WCDMA application which has been characterized by a significant variability of the processing effort. We proposed four flexible methods to schedule the Turbo decoding process on the processors of the platform which could trade off the quality of the results (services) and the required resources to produce the results. We utilized our structured performance model to derive and validate the proposed flexible methods for scheduling the Turbo decoding tasks. All proposed flexible scheduling (FS) methods in this project, when compared to a WCET scheduling method, improved the processors utilization. By using the *one shot* scheduling (first proposed FS method) comparing to the case of WCET, we could increase the number of users from 14 to 29 while keeping an acceptable quality of service reflected in degradation of 0.15 dB of decoder gain. Using

the *gradual* scheduling (second proposed FS method) allowed us to increase the number of users from 14 to 35 while keeping an acceptable quality of service reflected in a very small degradation of less than 0.1 dB of decoder gain. The *priority-driven one shot* scheduling (third proposed FS method) comparing to the *one shot* method provided the same results for increasing the number of users and average degradation of service quality while resulting the more uniform quality of services for different users. Also, the *priority-driven gradual* scheduling (fourth proposed FS method) compared to the *gradual* method provided the same results for increasing the number of users and average degradation of service quality while resulting in a more uniform quality of services for different users. Therefore, the *priority-driven gradual* scheduling is recognized as the most efficient method to allocate the resources to different users between the proposed FS methods.

Also, we structured a modified version of our performance model to validate our proposed method to model the Turbo decoding processes. Simulating such modified performance models and comparing the extracted results with the results of the corresponding performance models validated our Turbo decoding modeling method. In addition, we estimated the elapsed simulation times in two cases of functional and performance modelling for decoding the data blocks of 35 users in different channel conditions. We observed that the simulation times in the case of performance modelling were approximately 300 times faster in the case of functional modelling. Such ratio of simulation times justified the utilization of our performance modelling that allowed the rapid verification of different scheduling methods without performing detailed functional simulation.

The future work for this project consists of dynamic characterization of the other processing segments of an uplink WCDMA application in addition to the Turbo decoder and abstracting their execution time properties on the target processors. Also, data transmission between different processing segments of the application should be characterized in the form of data volumes and the elapsed time to transmit this data

between the processors of the platform. The result of such characterization could be introduced in our performance modelling strategy. Thus, different performance models could be created based on the developed modelling strategy to describe the different implementations of the whole application of an UMTS receiver base-station implemented on the target platform which represent all the processing segments of the application and all the data transmissions in the platform. Such complete models will allow to verify different methods of mapping and scheduling of the whole application on the multiprocessor platform and to study the maximum number of users which can be assigned to the base-station in different cases of implementations.

REFERENCES

[1] Atat, Y., Zergainoh, N.-E., *Simulink-based MPSoC design: new approach to bridge the gap between algorithm and architecture design*, VLSI. ISVLSI '07. IEEE Computer Society Annual Symposium on, 2007. pp. 9-14.

[2] Berrou, C., Glavieux, A., *Near Optimum Error Correcting Coding and Decoding: Turbo-Codes*, IEEE transactions on Communications, Vol. 44, 1996, pp. 1261-1271.

[3] Chen, Y. and Xiong, G., *Imprecise computation fault tolerant rate-monotonic scheduling*, in Proceedings of the 5th International Conference on Algorithms and Architectures for parallel Processing (ICA3PP '02), 2002, pp. 293–296.

[4] Cormier S., *Implementing a Turbo decoder for UMTS on a Vocallo multi-DSP device*, Technical Report, Ecole de Technologie Supérieur, 2007.

[5] Crozier, S., Gracie, K., Hunt, A., *Efficient Turbo decoding techniques*, In Proc. of Int. Conf. on Wireless Communications, 1999, pp. 187–195.

[6] Freeman, B. R., Statistically multiplexed Turbo code decoder, United States Patent, US 6,252,917 B1, Nortel Networks Limited, June 2001.

[7] Gao, K., Zhang, Y., He, S., and Gao, W., *Imprecise computation scheduling on scalable media stream delivery*, in Proceedings of ICICS-PCM 2003, vol. 3., 2003, pp. 1351-1355.

[8] Gilbert, F. , Thul, M. J., When, N., *Communication centric architectures for Turbo-decoding on embedded multiprocessors*, Proceedings of the conference on Design, Automation and Test in Europe, vol. 1, 2003, 10356p.

[9] Han, S.-I., Chae, S.-I., Brisolara, L., Carro, L., Reis, R., Guerin, X., Jerraya, A.-A., *Memory-efficient multithreaded code generation from Simulink for heterogeneous MPSoC*, Springer Netherlands, 2007, pp. 249-283.

[10] Hein, J. J. , Aylor, J. H. , Klenke, R. H., *Performance-based system design education*, In Proceedings of the IEEE International conference on Microelectronic Systems Education, 2003, pp. 35- 36.

[11] Holma, H., Toskala, A., *WCDMA for UMTS: Radio access for third generation mobile communications,* John Wiley & Sons (UK), 2003, 391 p.

[12] Ignat, N., Bélanger, N., Savaria, Y. and Nicolescu, G., *A MPSoC Architecture for Real-Time Systems with Significant Execution Time Variability*, Technical Report, École Polytechnique, 2008.

[13] Kempf, T. , Doerper, M. , Leupers, R. , Ascheid, G. , Meyr, H. , Kogel, T. , Vanthournout, B. , *A modular simulation framework for spatial and temporal task mapping onto multi-processor SoC platforms*, Proceedings of Design, Automation and Test in Europe, vol. 2, 2005, pp. 876- 881.

[14] Kim, K. H., Buyya, R. , and Kim, J., *Imprecise computation grid application model for flexible market-based resource allocation*, in Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID '06), 2006, vol. 1, pp. 5.

[15] Klenke, R. H., *Aylor, J. H., A proposed modeling environment to teach performance modeling and hardware/software codesign to senior undergraduates*, In Proceedings of the IEEE International Conference on Microelectronic Systems Education, 2003, pp. 27- 28.

[16] Liu, J. W. S., *Real-time systems*, Prentice Hall, 2000.

[17] Liu, W., *Efficient Application Mapping and Scheduling for Networks-on-Chip*, PHD thesis, Hong Kong University of science and technology, 2008.

[18] Malm, P., *Method for iterative decoder scheduling*, United States Patent, 7213189, ERICSSON TELEFON AB L M (SE), May 2007.

[19] Muller, O., Baghdadi, A., Jezequel, M., *ASIP-Based Multiprocessor SoC Design for Simple and Double Binary Turbo Decoding*, Design, Automation and Test in Europe, vol. 1, 2006, pp. 1-6.

[20] Neeb, C., Thul, M. J., When, N., *network-on-chip-centric approach to interleaving in high throughput channel decoders*, IEEE International Symposium on Circuits and Systems (ISCAS), 2005, pp. 1766-1769.

[21] Octasic Semiconductor Company Website, www.octasic.com, 2008.

[22] Oyamada, M., Wagner, F., Bonaciu, M., Cesario, W., Jerraya, A., *software performance estimation in MPSoC design,* 12th Asia and South Pacific Design Automation Conference, 2007, pp. 38-43.

[23] Rekh, S. , Rani, S.S., Shanmugam, A., *Optimal choice of interleaver for Turbo codes*, Academic Open Internet Journal, vol.15, part 6, 2005.

[24] Ristau, B., Limberg, T., Fettweis, G., *A mapping framework based on packing for design space exploration of heterogeneous MPSoCs*, Journal of signal processing systems, Springer New York, 2008.

[25] Streubuhr, M., Falk, J., Haubelt, Ch., Teich, J., Dorsch, R., Schlipf, Th., *Task-accurate performance modeling in systemC for real-time multi-processor architectures*, In Proceedings of Design, Automation and Test in Europe, 2006, pp. 480 – 481.

[26] Streubuhr, M., Jantsch, M., Haubelt, C., Teich, J., Schneider, A., *Semi-Automatic Generation of mixed Hardware/Software Prototypes from Simulink models*, Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen", Freiburg, 2008, pp. 139-148.

[27] *Universal Mobile Telecommunications System (UMTS) ; Physical layer - general description*, ETSI. 3GPP TS 25.201 version 6.1.0 Release 6, 2004.

[28] Valentin, M. C., Sun J., *The UMTS Turbo Code and an Efficient Decoder Implementation Suitable for Software-Defined Radio*", International Journal of Wireless Information Networks, Vol. 8, No. 4, 2001.

[29] Wiklund, D., Liu, D., *Design, mapping, and simulations of a 3G WCDMA/FDD basestation using network on chip*, Fifth international workshop on System-on-Chip for Real-Time Applications, 2005, pp. 252- 256.

[30] *3rd Generation Partnership Project; Specification Group Radio Access Network; Multiplexing and channel coding (FDD) (Release 7)*, 3GPP T TS 25.212 V7.1.0, 2006.

[31] *3rd Generation Partnership Project*, 3GPP web site, http://www.3gpp.org, 2008.

APPENDIX 1

Simulation results for the example described in section 3.3. The results correspond to 25 simulation stages.

```
INITIAL CASE:
rdy = [-1.000000 -1.000000 -1.000000 -1.000000 ]
st = [-1.000000 -1.000000 -1.000000 -1.000000 ]
t_sim = 0.000000
fin = [-1.000000 -1.000000 -1.000000 -1.000000 ]
load_unit = [0.000000 0.000000 ]


SIMULATION STAGE 1:
rdy = [0.000000 0.000000 -1.000000 -1.000000 ]
st = [0.000000 0.000000 -1.000000 -1.000000 ]
t_sim = 0.000000
EXECUTED TASKS: TASK 1
fin = [0.010000 -1.000000 -1.000000 -1.000000 ]
load_unit = [0.010000 0.000000 ]


SIMULATION STAGE 2:
rdy = [0.020000 0.000000 0.010000 -1.000000 ]
st = [0.020000 0.010000 0.010000 -1.000000 ]
t_sim = 0.010000
EXECUTED TASKS: TASK 2 TASK 3
fin = [-1.000000 0.020000 0.020000 -1.000000 ]
load_unit = [0.020000 0.020000 ]


SIMULATION STAGE 3:
rdy = [0.020000 0.020000 -1.000000 0.020000 ]
st = [0.020000 0.020000 -1.000000 0.020000 ]
t_sim = 0.020000
EXECUTED TASKS: TASK 1 TASK 4
fin = [0.030000 -1.000000 0.020000 0.030000 ]
load_unit = [0.030000 0.030000 ]


SIMULATION STAGE 4:
rdy = [0.040000 0.020000 0.030000 -1.000000 ]
st = [0.040000 0.030000 0.030000 -1.000000 ]
t_sim = 0.030000
EXECUTED TASKS: TASK 2 TASK 3
fin = [-1.000000 0.040000 0.040000 0.030000 ]
load_unit = [0.040000 0.040000 ]


SIMULATION STAGE 5:
rdy = [0.040000 0.040000 -1.000000 0.040000 ]
st = [0.040000 0.040000 -1.000000 0.040000 ]
t_sim = 0.040000
EXECUTED TASKS: TASK 1 TASK 4
fin = [0.050000 -1.000000 0.040000 0.050000 ]
load_unit = [0.050000 0.050000 ]
```

```
SIMULATION STAGE 6:
rdy = [0.060000 0.040000 0.050000 -1.000000 ]
st = [0.060000 0.050000 0.050000 -1.000000 ]
t_sim = 0.050000
EXECUTED TASKS: TASK 2 TASK 3
fin = [-1.000000 0.060000 0.060000 0.050000 ]
load_unit = [0.060000 0.060000 ]


SIMULATION STAGE 7:
rdy = [0.060000 0.060000 -1.000000 0.060000 ]
st = [0.060000 0.060000 -1.000000 0.060000 ]
t_sim = 0.060000
EXECUTED TASKS: TASK 1 TASK 4
fin = [0.070000 -1.000000 0.060000 0.070000 ]
load_unit = [0.070000 0.070000 ]


SIMULATION STAGE 8:
rdy = [0.080000 0.060000 0.070000 -1.000000 ]
st = [0.080000 0.070000 0.070000 -1.000000 ]
t_sim = 0.070000
EXECUTED TASKS: TASK 2 TASK 3
fin = [-1.000000 0.080000 0.080000 0.070000 ]
load_unit = [0.080000 0.080000 ]


SIMULATION STAGE 9:
rdy = [0.080000 0.080000 -1.000000 0.080000 ]
st = [0.080000 0.080000 -1.000000 0.080000 ]
t_sim = 0.080000
EXECUTED TASKS: TASK 1 TASK 4
fin = [0.090000 -1.000000 0.080000 0.090000 ]
load_unit = [0.090000 0.090000 ]


SIMULATION STAGE 10:
rdy = [0.100000 0.080000 0.090000 -1.000000 ]
st = [0.100000 0.090000 0.090000 -1.000000 ]
t_sim = 0.090000
EXECUTED TASKS: TASK 2 TASK 3
fin = [-1.000000 0.100000 0.100000 0.090000 ]
load_unit = [0.100000 0.100000 ]


SIMULATION STAGE 11:
rdy = [0.100000 0.100000 -1.000000 0.100000 ]
st = [0.100000 0.100000 -1.000000 0.100000 ]
t_sim = 0.100000
EXECUTED TASKS: TASK 4
fin = [-1.000000 -1.000000 0.100000 0.110000 ]
load_unit = [0.100000 0.110000 ]


SIMULATION STAGE 12:
rdy = [0.100000 0.100000 -1.000000 -1.000000 ]
st = [0.100000 0.100000 -1.000000 -1.000000 ]
```

```
t_sim = 0.100000
EXECUTED TASKS: TASK 1
fin = [0.110000 -1.000000 0.100000 0.110000 ]
load_unit = [0.110000 0.110000 ]


SIMULATION STAGE 13:
rdy = [0.120000 0.100000 0.110000 -1.000000 ]
st = [0.120000 0.110000 0.110000 -1.000000 ]
t_sim = 0.110000
EXECUTED TASKS: TASK 2 TASK 3
fin = [-1.000000 0.120000 0.120000 0.110000 ]
load_unit = [0.120000 0.120000 ]


SIMULATION STAGE 14:
rdy = [0.120000 0.120000 -1.000000 0.120000 ]
st = [0.120000 0.120000 -1.000000 0.120000 ]
t_sim = 0.120000
EXECUTED TASKS: TASK 1 TASK 4
fin = [0.130000 -1.000000 0.120000 0.130000 ]
load_unit = [0.130000 0.130000 ]


SIMULATION STAGE 15:
rdy = [0.140000 0.120000 0.130000 -1.000000 ]
st = [0.140000 0.130000 0.130000 -1.000000 ]
t_sim = 0.130000
EXECUTED TASKS: TASK 2 TASK 3
fin = [-1.000000 0.140000 0.140000 0.130000 ]
load_unit = [0.140000 0.140000 ]


SIMULATION STAGE 16:
rdy = [0.140000 0.140000 -1.000000 0.140000 ]
st = [0.140000 0.140000 -1.000000 0.140000 ]
t_sim = 0.140000
EXECUTED TASKS: TASK 1 TASK 4
fin = [0.150000 -1.000000 0.140000 0.150000 ]
load_unit = [0.150000 0.150000 ]


SIMULATION STAGE 17:
rdy = [0.160000 0.140000 0.150000 -1.000000 ]
st = [0.160000 0.150000 0.150000 -1.000000 ]
t_sim = 0.150000
EXECUTED TASKS: TASK 2 TASK 3
fin = [-1.000000 0.160000 0.160000 0.150000 ]
load_unit = [0.160000 0.160000 ]


SIMULATION STAGE 18:
rdy = [0.160000 0.160000 -1.000000 0.160000 ]
st = [0.160000 0.160000 -1.000000 0.160000 ]
t_sim = 0.160000
EXECUTED TASKS: TASK 1 TASK 4
fin = [0.170000 -1.000000 0.160000 0.170000 ]
load_unit = [0.170000 0.170000 ]
```

```
SIMULATION STAGE 19:
rdy = [0.180000 0.160000 0.170000 -1.000000 ]
st = [0.180000 0.170000 0.170000 -1.000000 ]
t_sim = 0.170000
EXECUTED TASKS: TASK 2 TASK 3
fin = [-1.000000 0.180000 0.180000 0.170000 ]
load_unit = [0.180000 0.180000 ]


SIMULATION STAGE 20:
rdy = [0.180000 0.180000 -1.000000 0.180000 ]
st = [0.180000 0.180000 -1.000000 0.180000 ]
t_sim = 0.180000
EXECUTED TASKS: TASK 1 TASK 4
fin = [0.190000 -1.000000 0.180000 0.190000 ]
load_unit = [0.190000 0.190000 ]


SIMULATION STAGE 21:
rdy = [0.200000 0.180000 0.190000 -1.000000 ]
st = [0.200000 0.190000 0.190000 -1.000000 ]
t_sim = 0.190000
EXECUTED TASKS: TASK 2 TASK 3
fin = [-1.000000 0.200000 0.200000 0.190000 ]
load_unit = [0.200000 0.200000 ]


SIMULATION STAGE 22:
rdy = [0.200000 0.200000 -1.000000 0.200000 ]
st = [0.200000 0.200000 -1.000000 0.200000 ]
t_sim = 0.200000
EXECUTED TASKS: TASK 1 TASK 4
fin = [0.210000 -1.000000 0.200000 0.210000 ]
load_unit = [0.210000 0.210000 ]


SIMULATION STAGE 23:
rdy = [0.220000 0.200000 0.210000 -1.000000 ]
st = [0.220000 0.210000 0.210000 -1.000000 ]
t_sim = 0.210000
EXECUTED TASKS: TASK 2 TASK 3
fin = [-1.000000 0.220000 0.220000 0.210000 ]
load_unit = [0.220000 0.220000 ]


SIMULATION STAGE 24:
rdy = [0.220000 0.220000 -1.000000 0.220000 ]
st = [0.220000 0.220000 -1.000000 0.220000 ]
t_sim = 0.220000
EXECUTED TASKS: TASK 1 TASK 4
fin = [0.230000 -1.000000 0.220000 0.230000 ]
load_unit = [0.230000 0.230000 ]


SIMULATION STAGE 25:
rdy = [0.240000 0.220000 0.230000 -1.000000 ]
st = [0.240000 0.230000 0.230000 -1.000000 ]
t_sim = 0.230000
```

```
EXECUTED TASKS: TASK 2 TASK 3
fin = [-1.000000 0.240000 0.240000 0.230000 ]
load_unit = [0.240000 0.240000 ]
```