Efficient scheduling of applications onto cloud FPGAs

Matteo Bertolino

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2021IPPAT001

Thèse de doctorat

TELECOM Paris

IP PARIS

Efficient Scheduling of Applications onto Cloud FPGAs

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom Paris

École doctorale n°626 Institut Polytechnique de Paris (IPP)
Spécialité de doctorat: Informatique

Thèse présentée et soutenue à Sophia Antipolis, le 22/01/2021, par

MATTEO BERTOLINO

Composition du Jury :

Robert De Simone INRIA Sophia Antipolis	Président
Liliana Cucu-Grosjean INRIA de Paris	Rapporteur
Frédéric Rousseau Université Grenoble Alpes	Rapporteur
Virginie Fresse Université Saint Etienne	Examinateur
Ludovic Apvrille Télécom Paris	Directeur de thèse
Renaud Pacalet Télécom Paris	Co-directeur de thèse
Andrea Enrici Nokia Bell Labs France	Co-encadrant de thèse

Résumé substantiel

Cette recherche doctorale a été réalisée en collaboration avec les laboratoires Nokia Bell Labs France et a été aussi financée par eux. Les Bell Labs sont connus par leur récente recherche dans le domaine de la télécommunication. Le domaine de recherche de cette thèse est une étude des algorithmes de programmation pour FPGAs. Cela est motivé par l'utilisation récente des accélérateurs comme FPGAs dans des infrastructures du centre de données dans le cloud pour mieux fournir les besoins de calcul du travail intensif. L'objectif de cette contribution est la minimisation du temps de latence des travaux pour les applications avec les dépendances de données internes lesquelles sont exécutées en cloud en louant l'utilisation de FPGAs car il peut être fait par d'autres ressources hardware (ex. : CPUs, stockage). Nous traitons les applications dont les tâches ne peuvent pas correspondre à la fois à la zone FPGA : ainsi, le FPGA peut être reconfiguré au moins une fois avant de compléter l'exécution de l'entière application. Le choix de quelles tâches nous devons attribuer à quel stage de reconfiguration, a un fort impact le temps de latence globale. Les algorithmes de planification efficaces, lesquels minimisent le temps de latence, sont intéressants tant pour les utilisateurs (le coût est lié au temps d'utilisation) que pour les fournisseurs (pour mieux utiliser le FPGA). Comme déjà vu dans le chapitre 3, la plupart des travaux existants sont basés sur des algorithmes lents et précis ou sur des heuristiques rapides dont la qualité n'est pas optimale. Dans ce manuscrit, on proposera une nouvelle solution de programmation dont la qualité est meilleure que les heuristiques courants tandis que le temps d'exécution est semblable à une de ces heuristiques (presque dizaine de millisecondes pour les applications communes). Il existe diffèrent ouvrages connexes qui capturent les applications et les architectures en utilisant des modèles trop abstraits pour trouver de bons (en termes de temps de latence) ou décisions valides de programmation. (ex : dans le Chapitre 3 nous verrons que le FPGA est souvent représenté avec un numéro unique qui indique la quantité de la logique reconfigurable). Pour cette raison, nous avons décidé de compter sur une ou plusieurs applications concrètes et modèles d'architectures dans notre contribution. En raison de la nécessité croissante pour exécuter des tâches de calcul intensif (ex. : apprentissage automatique, traitement de signal, cryptographie, etc.) pour lesquels l'exécution des software n'offre pas une performance suffisant, les architectures cloud équipées avec seulement le CPU ne sont pas plus suffisant. Une solution serait celle d'intégrer l'accélérateur de hardware, lequel inclut FPGAs et CPUs. En fait, il y a des différences entre eux et par conséquent ils ne sont pas complètement interchangeables. Ainsi, un FPGA peut être plus adapté à exécuter une application donnée, et réciproquement. Pour certains types de processus, FPGAs ont été démontré d'être capable d'assurer une meilleure efficacité énergétique que le GPUs. Dans ce document, les auteurs comparent l'exécution d'un Convolutional Neural Network (CNN) en un FPGA mais aussi

dans un GPU dans le système cloud de Microsoft Catapult. En effet, leur expérience ont démontré une efficacité énergétique d'un ordre de grandeur au bénéfice de l'exécution du FPGA.

Les applications pour lesquelles FPGAs sont le plus pratiques inclues les applications hautement parallèles et/ou avec des opérations élémentaires qui ne conviennent pas bien dans le modèles de programmation CPUs ou GPUS, par exemple, le changement de bit, les données personnalisés avec de largeurs de bits non standard, etc. Les canaux qui codent et décodent les algorithmes utilisés dans la télécommunication sont de bons exemples de ces applications. Dans quelques circonstances, l'apprentissage automatique et les algorithmes d'apprentissage profond peuvent être aussi de bons candidats, en particulier s'ils sont irréguliers et si les types de données utilisées sont optimisés aux représentations non standard. Un autre exemple est montré dans le chapitre 3, où le temps et l'efficacité énergétique meilleure de FPGAs sur le GPUs est démontrée par des applications de slinding-windows, appelées Sum of Absolute Differences, 2-D convolution et Correntropy. Les applications sliding-window sont une typologie spéciale de traitement de signaux numérique qui consiste à glisser un signal plus petit, appelée window (fenêtre), à travers différentes positions dans un signal plus grand. (ex, une image). À chaque position de la window (fenêtre), il y a souvent de nombreux calculs à exécuter. GPUs sont préférés pour les computations que peuvent exploiter leur parallélisme, quand les opérations concernent les opérations SIMD ou les opérations à virgule flottante, même certains FPGA récents intègrent aussi un grand nombre d'opérations à virgule flottante. Le rendu graphe est un exemple d'application dans laquelle le GPU est un ajustement naturel en raison du traitement à virgule flottante massivement parallèle. Les autres traitements de signal, l'apprentissage automatique ou les applications de l'apprentissage profond avec des caractéristiques très semblables sont fréquemment accélérés grâce à l'utilisation de GPUs. Les accélérateurs jouent un rôle fondamental dans le cloud computing. Dans cette thèse, nous nous concentrons sur les FPGAs. Comme nous verrons, FPGAs cloud sont gérés comme tout autre ressources informatique ou de stockage et loués aux utilisateurs finales par le fournisseur cloud, la plupart du temps à travers la virtualisation. Accordant avec le paradigme Software-as-a-Service (SaaS), les services fournis par un FPGA (ou par des FPGAs) sont accessibles par APIs. FPGAs peuvent être partagé et leurs usages est multiplexé entre les usagers de plusieurs manière: synchronisation, espacement etc. D'un point de vue de la synchronisation, les ressources de le hardware peuvent être efficacement utilisées par le fournisseur pour maximiser leur ROI (Return On Investment). La solution de programmation proposée dans ce manuscrit cible la minimisation du temps d'exécution d'une application (temps de latence), lequel est un avantage pour le prix payé par les utilisateurs. En effet, plus l'exécution du temps de latence est optimisée et moins de temps est loué à la ressource matériel (hardware). Dans le moderne data centers cloud les FPGAs sont architecturalement

organisés en groupes]. Les applications qui sont l'objectif de cette thèse contiennent un grand nombre de tâches dépendantes et potentiellement parallèles. Un autre exemple est l'agrégation de différentes applications avec le but de minimiser le temps de latence de l'ensemble des applications résultantes. Par exemple, celui est le cas du Spark streaming où le temps de latence de ce batch est le temps de fin de la dernière tâche. Les solutions existantes pour la programmation des applications peuvent être divisées en macro-familles. Avant tout nous avons des solutions qui se basent sur des formulations mathématiques exactes (ex. : programmation MILP). Elles assurent des solutions exactes au prix d'un temps d'exécutions potentiellement haut (jusqu'à heures, jours où années en accord avec la grandeur du problème). Cela est dû au très grand espace de solution qui caractérise le problème de programmation auquel nous nous attaquons. Les paramètres qui contribuent à la grandeur du problème sont inclus, mais ne sont pas limités à : dépendances entre les tâches, temps d'exécution, numéro des ressources demandés, temps de reconfiguration, caractéristiques du FPGA. Une autre famille bien connue est représentée par les heuristiques. Parmi les heuristiques, nous nous intéressons surtout aux list-based heuristiques. Dans les list-based heuristiques, dont les tâches individuelles sont triées dans une liste de priorité et assignées, en séquence, à la première unité disponible que correspond à leur demande de ressources. Les priorités peuvent être assignées de façon statique ou dynamique selon le différents caractéristiques, ex. : le temps d'exécution où occupation des ressources. Les list-based heuristiques ont deux avantages sur la proposition de cette thèse, appelé ce (i) ils travaillent même dans le cas où travail arrivent séquentiellement à être exécutés avec aucune connaissance préalable des emplois ultérieurs et (ii) ils sont plus vite en termes de temps d'exécution. Malheureusement, comme montré dans le chapitre 3, ces heuristiques calculent une solution de programmation qui est, en moyenne, pire que le temps de latence calculé avec notre approche. Les travaux basés sur les Meta-Heuristiques (MHs) tels que les Genetic Algorithms (GAs), Simulated Annealing (SA), Tabu Search (TS) et ainsi de suite sont très communs. En général, MHs commencent d'une solution initiale et explorent de façon itérative un sous-ensemble de l'espace de solution. Leur usage a un sens spécialement quand l'espace de solution est très grand pour être exploré entièrement, comme le problème de la planification statique des tâches sur les FPGAs. Les MHs peuvent être appliquées à une grande variété de problèmes. Au contraire, notre solution, comme montré dans le chapitre 3 traite explicitement les FPGAs et les hypothèses relatives aux tâches. En outre, ils fournissent des solutions de bonne qualité mais le temps de calcul est plus élevé que celui de notre contribution. Pour résumer, les solutions MH offrent une bonne solution qui nécessite de beaucoup de temps pour les calculs et les list-based heuristiques n'offrent pas ces bonnes solutions de façon rapide. Comme le montre notre manuscrit, Slot effectué de manière rapide et trouve une solution meilleure que les list-based heuristiques. Dans cette thèse nous mettons l'accent sur le scheduling

d' application sur le FPGAs. Nous prenons comme hypothèse que les applications sont composées de tâche dépendant (exprimable comme DAGs). Ainsi nous supposons que la somme des ressources matérielles pour chaque tâche à exécuter dans le FPGA excède les ressources matérielles de le FPGA. Par conséquent, nous ciblons les applications qui nécessitent au moins deux stages de reconfigurations à exécuter. Chaque stage de reconfiguration est ainsi composé d'une reconfiguration totale du FPGA suivi par un sous-ensemble de tâches de l'application. Les stages de reconfiguration sont exécutés séquentiellement et la séquence des stages de reconfiguration doit respecter la dépendances de données de le DAG. En d'autres mots, si la tâche B dépend de la tâche A, la tâche A ne peut pas faire partie du stage de reconfiguration qui suit le stage de reconfiguration qui inclut la tâche B. Si la tâche A et la tâche B font partie du même stage de reconfiguration, alors A doit être totalement exécuté avant que B commence à être exécuté. Évidemment, la construction des stages de reconfiguration influence fortement l'application du temps de latence. Conséquemment, notre problème de programmation consiste dans l'identification rapide des stages de reconfiguration qui mènent à l'exécution d'une application avec un temps de latence lequel est proche ou égale au temps de latence optimale. La programmation sur FPGAs est diverse de la programmation sur les CPUs. En effet, l'implémentation hardware d'une tâche exige des ressources comme des éléments logiques reconfigurables, blocs de mémoire intégrés, etc. Ces ressources peuvent rassembler à des transformateurs, ex. : mémoires, registres, noyaux etc. En plus, les ressources hardware du CPUs et des FPGAs ne sont pas suffisantes pour exécuter toutes les tâches d'une application. Néanmoins, passer d'une tâche à une autre dans les CPUs est souvent plus rapide que reconfigurer un FPGA. Ainsi, reconfigurer un FPGA peut introduire à un chronométrage qui peut être comparé à le HET de la tâche. Par conséquent, le choix de quelles tâches il faut exécuter ensemble dans un stage de reconfiguration a un grand impact pour les FPGAs que des choix semblables pour le transformateur. En plus, à cause de cette surcharge, les reconfigurations sur le FPGAs devraient être moins fréquentes du changement de contexte sur le processeur. Le problème de programmation que nous ciblons dans cette thèse est semblable à la catégorie de Resource-Constrained Scheduling Problem (RCSP). Informellement un RCSP considère les ressources et les activités limitées à une durée connue. Un RCSP prévoit aussi que les demandes de ressources soient liées par des relations précédentes. Un RCSP consiste à finding a schedule of minimal duration en assignant un temps de départ à chaque activité ainsi les relations précédentes et les ressources disponibles sont respectées. Les travaux cités dans le chapitre 3 démontrent que le RCSP classique est un grand problème NP-hard. Néanmoins, notre problème de recherche diffère du RCSP classique parce que la reconfiguration FPGA présente une nouvelle variable dans l'espace de solution dont le surcharge contribue considérablement au temps de latence totale. En considérant différents exemples, on peut remarquer les intuitions suivantes : (i) la

meilleure solution n'est toujours celle qui possède un plus petit nombre de reconfigurations et (ii) la mesure clé est la quantité de temps utilisé pour exécuter les tâches de façon parallèle et qui peuvent être aussi maximisés. Afin de le faire, les tâches qui consomment plus de temps doivent être probablement considérées comme les plus importantes parce que elles ont des avantages potentiellement plus grands.

Notre solution essaye de minimiser le temps de latence d'une application en regroupant les tâches en stages de reconfigurations. Comme montré dans les sections précédentes, le parallélisme entre tâches et tâche dominantes (ex. : tâches qui ont des HET plus élevés) sont les points clé des problèmes qui sont traités par nos solutions. En effet, notre solution est liée à la sélection des tâches dominantes et nous voulons aussi mettre ces tâches dominantes en parallèle avec un sous-ensemble de tâches qui peuvent être exécutés en parallèle. Afin de sélectionner quelles tâches mettre en parallèle à la tâche dominante, nous considérons tout le graphe, et pas seulement la qualité du parallélisme entre la tâche dominante et la tâche parallèle à elle. Ainsi, notre décision considère les ressources demandées et les HET des tâches, les dépendances des données entre eux, les caractéristiques du FPGA et le temps de reconfiguration. Parmi toutes les tâches possibles qui peuvent être mises en parallèle avec la tâche dominante, nous définissons, dans ce travail, un approche de base qu'aide à définir quelles sont les tâches à sélectionner.

Une fois que la tâche dominante et les tâches mises en parallèle ont été sélectionnées grâce au service de notation (ex. : la création d'un stage de reconfiguration), le graphe a été remanié pour fusionner toutes les tâches qui ont été sélectionnées dans cette itération dans un nœud unique. À chaque itération, il se crée un nouveau stage de reconfiguration et ses tâches se fussent ensemble dans un seul nœud du graphe. L'input du graphe devient une séquence de stages de reconfiguration. Cette séquence exprime la programmation sélectionnée. Pour rappel, un stage de reconfiguration contient la reconfiguration totale du FPGA suivi par l'exécution des tâches qui appartiennent aux stages de reconfiguration. Comme le montre le chapitre 4 un processus d'optimisation finale essaie de compacter davantage les stages de reconfiguration afin de continuer à réduire le temps de latence. Notre approche mire à identifier, très probablement, une bonne solution, que va minimiser le temps de latence, parce que au lieu de considérer seulement une partie donnée du graphe comme quand commence de la tâche initiale, nous préférons considérer la tâche dominante (quelle que soit son emplacement dans le graphe) et le parallélisme plus puissant du graphe pour mieux exploiter la capacité de parallélisme des FPGAs. Cet heuristique est présenté dans le Chapitre 4, tandis que l'efficience de cet heuristique est présenté dans le Chapitre 5. Les FPGAs modernes ne sont pas simplement un récipient de hardware reconfigurable mais ils peuvent aussi incorporer d'autres éléments comme les processeurs généraux, les DSPs et ainsi de suite. L'un des pionniers de ce type hétérogène de hardware reconfigurable par un software est le dispositif Xilinx Virtex II Pro, où le FPGA inclut un IBM PowerPC405. De ce moment,

ont été apportées différentes améliorations. Par exemple, les produits Stellarton distribuent un processeur Intel Atom E6XX avec un FPGA Intel dans le même paquet. La dernière tendance pour le calcul d'haute performance est représentée par des projets comme Cygnus, un superordinateur qui a été développé près le Center for Computational Sciences (CCS), en Tsukuba, lequel intègre un mélange de CPUs, GPUs et FPGAs. Le calcul intensif n'est pas le seul domaine dans lequel les processeurs et les accélérateurs travaillent ensemble, parce qu'il est très commun dans les infrastructures cloud. Le problème de programmation que nous ciblons dans cette thèse est similaire à la catégorie de Resource-Constrained Scheduling Problem (RCSP). Les RCSP sont des problèmes programmés dont la programmation est influencée par la disponibilité ou le manque de ressources. Cela signifie, souvent, qu'en raison de la limitation des ressources, seulement une certaine application prendra plus de temps. Le chapitre 3 définit généralement le RCSP et présente la manière dont notre problème de programmation spécifique du FPGA s'adapte à cette classification. Le reste du travail est composé par deux aspects différents. On analyse comme chaque entité (ex. : application et architecture) est modelée avec la contribution du programme FPGA. En d'autres mots, on présente un aperçu concernant les inputs, les outputs, les hypothèses, les modèles et les paramètres nécessaires pour calculer le programme d'application dans les FPGAs. Les modèles sont utilisés pour décrire les applications et aussi l'architecture. En ce qui concerne les applications, nous mettons l'accent sur la présentation des assets de modélisation lesquels permettent de capturer les paramètres d'input pertinent pour résoudre le problème de programmation. Les modèles utilisés dans les problèmes de la programmation ont généralement beaucoup plus de paramètres grossières que les modèles utilisés par les travaux dont l'objectif est, par exemple, de générer une implémentation spécifique de hardware FPGA pour une application bien spécifique. Il est possible de conduire le même discours pour les architectures : comme nous verrons, les FPGAs sont caractérisés par une architecture complexe et les détails ne sont pas tous pertinents à la programmation. Les stratégies de programmation peuvent, en effet, faire une abstraction des paramètres avec un impact négligeable sur le temps d'exécution de l'algorithme de programmation. À cet égard, le chapitre 3 décrit comme les travaux connexes modèlent les architectures FPGA. Les modèles imprécis (ex. : les modèles qui ne considèrent pas les détails pertinents à la programmation) pourraient mener à des solutions de programmation sous-optimales ou erronées.

Modeler est important mais si nous faisons une comparaison avec la programmation, l'algorithme utilisé pour résoudre le problème de programmation est également important. Dans ce qui suit, nous allons désigner l'algorithme de résolution comme la « stratégie de programmation ». Ainsi, on va illustrerles œuvres principales de l'art qui traitent les stratégies de programmation et cela représente la partie la plus large de ce chapitre. Nous avons classifié les travaux principaux basés sur la stratégie

de programmation qui l'utilisent. En général, une stratégie de programmation peut être basée sur les meta-heuristique (MHs) ou sur les heuristiques. En outre, nous classons les derniers en list-based heuristiques et les packing-based heuristiques. Nous préférons nous concentrer seulement sur certaines de ces catégories pour les solutions d'haute qualité qu'elles fournissent (ex. : meta-heuristiques, formulations exactes) ou pour le temps d'exécution très rapide (ex. : list-based algorithmes de programmation). La contribution de cette thèse peut être placée dans la catégorie "packing-based" (ex. : les algorithmes qui prennent des décisions considérant les groupes des tâches. Cela permet d'atteindre des résultats qui peuvent se comparer avec les meta-heuristiques en termes de qualité tandis que maintenir le temps d'exécution peut être comparable aux list-based heuristiques. Pour ne pas être incomplète, les formulations mathématiques exactes peuvent être utilisées comme spécification du problème également par les heuristiques non exactes.

Afin d'évaluer la qualité du planning calculé par le Slot, nous devons comparer le temps de latence obtenu avec une référence optimale absolue. Donc, en premier lieu, nous avons formellement modelé le problème de programmation du FPGA et nous l'utilisons pour résoudre ces cas problématiques avec une approche exacte. Nous choisissons d'utiliser le Mixed Integer Linear Programming (MILP) parce qu'il semble vraiment adapté pour ces types de problèmes d'optimisation. Puis nous désignons un générateur d'instance aléatoire pour générer un grande nombre d'instances du problème de programmation du FPGA. Nous résoudrons ces instances avec un solveur MILP pour obtenir un temps de latence mineur et le comparer avec le temps de latence trouvé par le Slot dans les mêmes cas. Nous comparons aussi le temps d'exécution des deux approches. Enfin, nous comparons la qualité et les temps d'exécution du Slot avec le HEFT-NF heuristique dont nous avons déjà discuté. Au meilleur de notre connaissance HEFT-NF est la seule proposition comparable au Slot.

Dans le chapitre sur l'évaluation, nous avons montré parce qu'une analyse exhaustive est impossible du point de vue du temps de calcul. Puis, nous avons décrit comment nous capturons les ressources hardware et les applications. Enfin nous avons décrit les étapes principales du Slot et nous avons discuté sa complexité théorique. Nous pensons que le clé-force du Slot est son calcul intelligent des stages de reconfiguration lesquels exploitent efficacement le parallélisme d'un FPGA. Le chapitre suivant vous montrera l'évaluation d'un Slot sur un indice de référence synthétique composé par différentes instances générées de manière pseudo-aléatoire. En particulier, nous allons évaluer la qualité de la solution de programmation produite par le Slot et son temps d'exécution.

Nous comptons sur le fait que le Slot pourrait être également intéressant pour le design du système intégré complexe. Ainsi, dans ce chapitre, nous appliquons le Slot dans un contexte différent que les centres de données cloud, comme Model-Driven Engineering. Model-Driven Engineering (haut niveau) modèle le système

intégré en offrant des modèles dédiés pour capturer des composants hétérogènes de hardware/software. Les modèles peuvent représenter une application, une plateforme et donc la manière dans laquelle une application peut être mappée sur une plateforme. En plus, les modèles peuvent être transformés afin de générer de modèles exécutables (pour des vérifications formelles ou pour des fins de simulation) et un code exécutable pour les modèles de haut niveau. Grâce à leurs abstractions internes, les modèles devraient aider à se concentrer sur les aspects les plus importants du système. À l'égard des FPGAs, nous pensons que ces deux caractéristiques importantes devraient être prises en considération : parallélisme de hardware et reconfiguration dynamique.

Pour mieux supporter le design des systèmes intégrés, nous avons intégré un Slot dans un cadre de Model-Driven Engineering (MDE) appelé TTool. TTool est un instrument gratuit et open-source qui supporte différents stages de développement avec UML/SysML (ex. : déterminations des besoins, analyses, partitioning hardware/software et design software (intégré). Parmi les différents instruments MDE, TTool a été sélectionné en raison de sa légèreté et de son extension facile, TTool, en effet, a déjà démontré de supporter les applications de traitement de signaux. Nous avons intégré le Slot aux TTool/DIPLODOCUS à travers un plugin. TTool/DIPLODOCUS étaient déjà capables de représenter les FPGAs et les reconfigurations dynamiques pour effectuer des simulations sur les tâches mappés sur le FPGA. Encore, la programmation des tâches pour les FPGAs devrait être faite à main. Étant donné une application mappée sur le FPGA, le Slot peut être appliqué pour déterminer les stages de reconfiguration de la programmation. Cet information de programmation peut être transmise au moteur de simulation – par exemple pour vérifier que cette programmation sélectionnée travaille bien avec les autres parties du système – où qu'elle peut être utilisée par le moteur Design Space Exploration qui peut utiliser l'information fournit par un stage de reconfiguration pour prendre des décisions davantage.

Le travail futur de cette thèse cible à des architectures différentes des celles présentées dans le Chapitre 7. En effet, nous voulons appliquer les principes du Slot aussi dans d'autres typologies de situation qui pourraient se vérifier dans les projets réels. Nous rappelons au lecteur que les stages principaux du Slot sont (i) privilégier les tâches dominantes, (ii) la génération d'un ensemble de stages de reconfiguration candidats commence par la tâche dominante, (iii) l'évaluation du stage de reconfiguration plus promettant à l'égard du temps de latence globale (pour cela nous utilisons un système de notation) et (iv) optimiser davantage la solution en compactant les stages de reconfiguration sélectionnés. Grâce à l'approche modulaire du Slot, nous comptons qu'avec une légère adaptation les différents stages peuvent utiliser le Slot pour d'autres problèmes de recherche.

Ainsi nous proposons les six directions suivantes:

Pour résoudre les problèmes d'évolutivité du Slot: de la complexité de la discus-

sion et l'évaluation, nous avons remarqué que le Slot ne s'adapte pas très bien avec le nombre des stages de reconfiguration candidats, même si les dépendances des données et les ressources disponibles limitent le nombre des stages de reconfiguration candidats. Cela peut affecter le temps d'exécution du heuristique dans le cas où il est appliqué aux graphes avec des centaines de tâches.

La programmation des tâches que peut requêter des ressources programmées dépendantes : nous avons classifié les ressources en deux typologies : appelé scheduling-independent et scheduling-dependent resources. Une scheduling-independent resource est une ressource qu'est assignée exclusivement à une tâche pour la durée totale du stage de reconfiguration et que contient la tâche. Ainsi, une requête pour une spécifique scheduling-independent resource (tels que LEs) sans un stage de reconfiguration peut être géré simplement en ajoutant les deux ensembles. Une scheduling-dependent resource est une ressource qui peut être assignée à une tâche pour la durée totale de la tâche. La dernière est toujours mineure ou égale à la vie totale du stage de reconfiguration qui contient la tâche. Les tâches qui enquêtent une scheduling-dependent resource excèdent la limite physique pour un FPGA spécifique qui peut faire partie du même stage de reconfiguration comme s'ils étaient scheduling-independent ainsi, probablement de façon pessimiste.

Une discussion intéressante est que le Slot travaille mieux quand toutes les tâches qui composent les graphes d'application diffèrent l'un de l'autre. En effet, dans le cas où le graphe d'application contient des multiples instances de la même tâche mais dans différent parties du graphe, nous pensons que la meilleure manière pour calculer les stages de reconfiguration devrait être bien définie. En plus, dans le Slot future il faut considérer des multiples implémentations pour la même tâche et décider quel peut apporter plus bénéfices à la réduction du temps de latence.

Le Slot courant est appliqué à une ou plusieurs applications qui sont exécutées sur un seul FPGA. En future, nous planifions de l'appliquer à des meilleures architectures ciblées lesquelles permettent d'exploiter les capacités des FPGAs à distance, comme dans Microsoft Catapult. Si nous supposons que cette latence est vraiment basse au regard du HET des tâches et au regard de la reconfiguration du FPGA, une solution pourrait simplement être celle d'adapter le Slot courant en résumant ensemble les ressources des FPGAs à distance.

Enfin, nous aimerions mieux adapter le Slot pour mieux aborder les changements dynamiques que le contexte des centres de données cloud peut nécessiter. En particulier, nous voulons relancer de façon dynamique le Slot chaque fois que la nouvelle application est assignée à un FPGA qui est déjà en train d'exécuter une application.

Ce manuscrit présente la programmation des applications sur les FPGAs. Nous avons supposé que les applications sont composées par des tâches dépendantes et les FPGAs soient totalement reconfigurés. Notre but est de minimiser le temps de latence de l'application. Nous avons placé notre contribution dans les centres de

données cloud. Nous montrons comment les architectures du centre de données cloud intègrent les accélérateurs de le hardware, comme les FPGAs, pour mieux supporter les besoins croissants des applications en termes de puissance de calcul. Le travail connexe montre que toutes ces approches existantes ne peuvent pas répondre de façon efficace à notre problématique. Nous présentons notre approche, Slot, laquelle offre un juste compromis entre la qualité de la solution (en termes de temps de latence) et le temps d'exécution qu'elle nécessite pour identifier cette solution. Le Slot est basé sur un processus itératif. Premièrement, cela considère tous les graphes et la tâche dominante de ce graphe. Dès les stages de reconfiguration qui peuvent être construits à partir de ce graphe, nous utilisons une méthode de notation qui permet de sélectionner le meilleur stage de reconfiguration. Quand un stage de reconfiguration a été sélectionné, nous fusionnons toutes les tâches de ce stage de reconfiguration ensemble dans le graphe. Enfin, cette itération extrants une séquence de stages de reconfiguration laquelle, après une optimisation finale, représentera la programmation proposée.

Nous avons montré que le Slot est efficace sur un benchmark de 37500 graphes générés de manière pseudo-aléatoire. Nous l'avons aussi comparé aux autres deux approches : Une formulation MILP, laquelle retourne souvent une solution optimale, et un heuristique existant lequel a été adapté pour mieux cibler le problème de la programmation du FPGA (appelé HEFT-NF). Dans la partie finale du manuscrit, nous montrons que le Slot peut être appliqué dans des contextes différents du centre de données cloud. Nous allons montrer comme nous l'avons intégré dans un outil d'ingénierie pilote, appelé TTool/DIPLODOCUS, lequel supporte la conception initiale des systèmes embarqués.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

This doctoral research has been realized in collaboration with Nokia Bell Labs France laboratories and has been financed by it. Bell Labs are known for their recent research in the telecommunication domain. The research domain of this thesis is a study of scheduling algorithms for FPGAs. This is motivated by the recent use of accelerators such as FPGAs in cloud data center infrastructures to better supply the computation needs of intensive workloads. The target of this contribution is the makespan minimization for applications with internal data-dependencies, which are executed in cloud by renting the use of FPGAs as it can be done for other hardware resources (e.g., CPUs, storage). We address applications whose tasks cannot fit at once the FPGA area: thus, the FPGA must be reconfigured at least one time before completing the execution of the entire application. The choice of which tasks are assigned to which slot has a strong impact on the overall makespan. Efficient scheduling algorithms which minimize the makespan are interesting both for the users (cost is related to usage time) and for providers (to better utilize the FPGA pool).

As discussed in Chapter 3, the vast majority of existing works is either based on slow and precise algorithms or on fast heuristics whose quality is not close to the optimum. In this manuscript, we will propose a new scheduling solution whose quality is better than the current heuristics whila having run-time which is similar to the one of these heuristics (around tens of milliseconds for common applications). Several related works capture applications and architectures using models which in our opinion are too abstract to find good (in terms of makespan) or valid scheduling decisions. (e.g., we will see in Chapter 3 that the FPGA is often represented with a single number that indicates the amount of reconfigurable logic). For this reason, we have decided to rely on more concrete application and architecture models in our contribution.

1.2 Motivation

Because of the increasing necessity to execute computationally-intensive tasks (e.g., machine learning, signal processing, cryptography, etc.), for which software execution does not offer sufficient performance, cloud architectures equipped with only CPUs are not sufficient anymore. A solution is to integrate hardware accelerators, which include FPGAs and GPUs. Actually, there are important differences between them and thus they are not completely interchangeable. As a consequence, an FPGA can be most suitable to execute a given application, and reciprocally. For certain types of processing, FPGAs have been proved to be able to ensure a better energy efficiency than GPUs, as shown in [89]. In this paper, authors compare the execution of a Convolutional Neural Network (CNN) both in an FPGA and in a GPU in the Microsoft Catapult cloud system. Indeed, their experiments have shown an energy efficiency of one order of magnitude in favor of FPGA execution.

Applications for which FPGAs are the most convenient include highly parallel applications and/or with elementary operations that do not fit well in CPUs or GPUs programming models, for instance bit swapping, custom data types with non-standard bit widths, etc. Channel coding and decoding algorithms used in telecommunication are good examples of such applications [56] [99]. In certain circumstances machine learning and deep learning algorithms can also be good candidates, especially if they are irregular and if the used data types are optimized to non-standard representations.

Another example is shown in paper [50], where the better temporal and energy efficiency of FPGAs over GPUs is demonstrated for sliding-windows applications, namely Sum of Absolute Differences, 2-D convolution and Correntropy. Sliding-window applications are a special type of digital signal processing that consist in sliding a smaller signal, named window, across different positions in a larger signal (e.g., an image). At each window position, there is usually a computationally intensive function to execute.

GPUs are preferred for computations which can exploit their parallelism, when operations concern SIMD operations or floating point operations, even some recent FPGAs also embed large numbers of floating point units. Graphics rendering is a typical example of application where GPUs are a natural fit because of its massively parallel vector floating point processing. Other signal processing, machine learning or deep learning applications with similar characteristics are also frequently accelerated using GPUs.

Accelerators play a crucial role in Cloud computing. In this thesis, we focus on FPGAs. As we will in Section 2.3, cloud FPGAs are managed like any other computing or storage resources and rented to final users by cloud providers, most of the time through virtualization. According with the paradigm Software-as-a-Service (SaaS), services provided by an FPGA (or by a pool of FPGAs) are accessed through APIs.

FPGAs can then be shared and their usage is multiplexed among users in various ways: timing, spacing, etc.

From a timing point of view, hardware resources must be efficiently utilized by the provider to maximize its ROI (Return On Investment). The scheduling solution proposed in this manuscript targets the minimization of the execution time of an application (makespan), which is an advantage for the price users pay. Indeed, the more the execution makespan is optimized, the less time is rented the hardware resource.

In modern cloud data centers FPGAs are architecturally organized in a pool [91]. Applications that are the target of this thesis contain a large number of dependent and potentially parallel tasks. Another example is the aggregation of different applications with the objective of minimizing the overall makespan of the whole resulting application. For instance, this is the case of Spark Streaming [111], where makespan of this batch is the termination time of the last task [112].

Existing solutions for the static scheduling of applications can be divided into macro-families. First, we have solutions based on exact mathematical formulations (e.g., MILP programming). These ensure exact solutions at the price of potentially high execution times (up to hours, days or years according to the size of the problem). This is due to the very large solution space which characterizes the scheduling problem we are addressing. Parameters that contribute to the size of the problem include, but are not limited to: dependencies among tasks, execution times, number of requested resources, nature of requested resources, reconfiguration time, FPGA features.

Another well-known family is represented by heuristics. Among heuristics, we will particularly deepen list-based heuristics. In list-based heuristics, which individual tasks are sorted in a priority list and assigned, in sequence, to the earliest available unit that fits their resource request. Priorities can be assigned statically or dynamically according to different characteristics, e.g., execution time or resource occupancy. List-based heuristics have two advantages over the proposal of this thesis, namely that (i) they work even in the case where jobs sequentially arrive to be executed with no prior knowledge of subsequent jobs and (ii) they are faster in terms of run-time. Unfortunately, as shown in Chapter 5, these heuristics compute a scheduling solution which is, in average, worse than the makespan computed by our approach.

Works based on Meta-Heuristics (MHs) such as Genetic Algorithms (GAs), Simulated Annealing (SA), Tabu Search (TS) and so on are also very common. In general, MHs start from an initial solution and iteratively explore a subset of the solution space. Their usage has a sense especially when the solution space is too large to be entirely explored, such as the problem of static scheduling of tasks onto FPGAs. MHs can be applied to a wide variety of problems. On the contrary, our solution as shown in Section 4.2, explicitly handle FPGAs and tasks related assumptions. More-

over, they provide good quality solutions but the computation time that is higher than the one of our contribution [93].

As a summary, exact and MH solutions offer a good solution but which take too much time to compute and list-based heuristics offer not that good solutions in a fast way. As shown in our manuscript, *Slot* performs in a fast way and find a solution which is better than list-based heuristics.

1.3 Glossary

Following a list of most common concepts we use throughout the thesis. Some of these concepts may have more interpretations, so we state here which meaning they have within this manuscript.

- **Scheduling:** we define the scheduling of an application onto a reconfigurable device the process to select an execution order among dependent tasks.
  - **On-line/Off-line scheduling of applications:** on-line scheduling consists in making scheduling decisions during the execution of the applications while off-line scheduling is precomputed before the applications executes. The choice between on-line and off-line scheduling is frequently guided by criteria like the frequency at which scheduling decisions must be made, the run-time of the tasks to schedule, the run-time of the scheduling algorithm itself, the determinism of the applications...
- **Makespan:** the total duration between the start and the end of execution of an application.
- **Reconfiguration of an FPGA**
  - **Full** reconfiguration: it consists in saving the logical setting of an FPGA by using a bitstream.
  - **Partial** reconfiguration: it allows a limited, predefined portion of an FPGA to be reconfigured while the configuration of the remainder of the device is unmodified
- **FPGA scheduling problem:** the problem of how to decide in which order the tasks of an application or of a set of applications shall be executed on an FPGA. These decisions must usually meet requirements (e.g. the inter-task dependencies or the instantaneous amount of available resources) and try to optimize one or more specific objectives (e.g. energy consumption or total makespan). The way scheduling decisions are taken also depends on the context (partial or total FPGA reconfiguration, on-line or off-line scheduling, etc.).

In this contribution we consider total FPGA reconfiguration and our objective is the minimization of the total makespan.

- **Slot:** it has a dual meaning
  - If ***Slot*** is written in italic and with the uppercase S - it refers to the name of the algorithm presented in this thesis.
  - Else, if **slot** is written lowercase and not italicized - it consists in a total reconfiguration of the FPGA, followed by the execution of the tasks in the configuration, and the release of the device. A slot usually includes one or more tasks whose (a part of) processing time (HET, explained later in this list) may overlap. In the rest of this thesis, we will use the following notation for a slot: $[R,\ \{t_0\ \ldots\ t_n\}]$, where:
    - "$[$" denotes the beginning of a slot;
    - "R" represents a total FPGA reconfiguration;
    - "$\{t_0\ \ldots\ t_n\}$" are the set of tasks which compose the slot. Tasks $t_0 \ldots t_n$ are not executed sequentially, but they follow the dependencies of the application;
    - "$]$" denotes the end of a slot.
- **Hardware Execution Time - HET:**
  - **Task Hardware Execution Time** - task HET: the processing time of a task on the FPGA.
  - **Slot Hardware Execution Time** - slot HET: the total time taken by a slot.
- **Valid solution:** a scheduling solution is composed of a sequence of slots. In our FPGA scheduling problem, such solution is valid if it respects two conditions. First, tasks within a slot must not exceed the capabilities of the FPGA in terms of resources. Second, the sequence of slots must respect the data dependencies between tasks. This means that a task of slot N cannot be a successor of a task of slot $N + k$ ($k > 0$).
- **Scheduling-independent resource:** a scheduling-independent resource is a resource which is exclusively assigned to a task for the entire lifetime of the slot that contains the task.
- **Scheduling-dependent resources:** a scheduling-dependent resource is a resource which is assigned to a task for the entire task's lifetime. The latter is always less or equal the entire lifetime of the slot that contains the task.

1.4 Research Problem

In this thesis we focus on the **scheduling of applications** onto FPGAs. We take the assumptions that applications are composed of **dependent tasks** (expressible as **DAGs**). We thus assume that the sum of the physical resources necessary for each task to execute in the FPGA exceeds the physical resources of the FPGA. As a consequence, we target applications which need at least two slots to execute. Each slot is thus composed of a total reconfiguration of the FPGA followed by a subset of tasks of the application. Slots are executed sequentially and the sequence of slots must respect data dependencies of the DAG. In other words, if a task B depends on a task A, task A cannot be part of a slot that follows a slot that includes task B. If tasks A and B are part of the same slot, then A must be totally executed before B can start being executed. Obviously, the construction of slots strongly influence the application makespan. Consequently, **our scheduling problem consists in identifying in an fast way slots that lead to execution an application with a makespan which is close or equal to the optimum makespan.**

Scheduling onto FPGAs is different than scheduling on CPUs. Indeed, a hardware implementation on FPGA of a task requires resources such as reconfigurable logic elements, embedded memory blocks, etc. These resources may look like the one of processors, e.g., memory, registers, cores, etc. Also, hardware resources of both CPUs and FPGAs may not be sufficient to execute all together all the tasks of an application. However, switching from one task to another in CPUs is usually much faster than reconfiguring an FPGA [107]. Thus, reconfiguring an FPGA may introduce a timing overhead comparable to the HET of a task. As a consequence, the choice of which tasks execute together in a slot has a larger impact for FPGAs than similar choices for processors. Additionally, because of this overhead, reconfigurations on FPGAs are expected to be less frequent than context-switching on processors.

The scheduling problem we target in this thesis is similar to Resource-Constrained Scheduling Problem (RCSP) category. Informally, a RCSP considers limited resources and activities of known durations. A RCSP also expects resource requests to be linked by precedence relations. A RCSP consists in **finding a schedule of minimal duration** by assigning a start time to each activity such that the precedence relations and the resource availabilities are respected [101]. For a taxonomy of RCSPs, we invite the reader to consult the work in [60]. The authors in [30] demonstrated that the classical RCSP is a strong NP-hard problem.

However, our research problem differs from the classical RCSP because FPGA reconfigurations introduce a new variable in the solution space whose overhead significantly contributes to the overall makespan. As said before, choices of tasks that compose a slot has a strong impact on the overall makespan. Let us for instance consider the example in Figure 1.1. In this figure *Source* and *Sink* are artificial tasks,

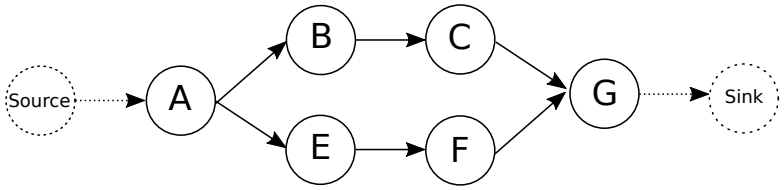


Figure 1.1: 8-tasks DAG example

Task	Occupancy	Time
A	33%	10 [u]
B	33%	300 [u]
C	33%	20 [u]
E	33%	150 [u]
F	33%	150 [u]
G	33%	10 [u]

Table 1.1: Parameters of tasks for the example of Figure 1.1 - Occupacy represents the percentage of the hardware resources used by tasks in a given FPGA and the time represents the number of time units that the tasks take to execute in the given FPGA

whereas other tasks are parametrised with hardware occupancy and HET; as listed in Table 1.1. For simplicity reasons, each task of this example occupies 33% of the FPGA. This means that up to three tasks can be put in the same slot.

We consider an FPGA whose reconfiguration R requires $T_R = 40$ units of time. We now compare two possible schedulings:

- **Scheduling** 1**:** [R, {A B E}], [R, {C F G}] $\rightarrow$ 550 [u]
- **Scheduling** 2**:** [R, {A}], [R, {B E F}], [R, {C G}] $\rightarrow$ 460 [u]

Scheduling 1 minimizes the number of reconfigurations. Since up to three tasks can share the FPGA area, A, B and E are put in the same slot. The next slot includes tasks C, F and G. The overall makespan is therefore 550 units of time, included 80 units spent in reconfigurations. Scheduling 2 is less intuitive. Indeed, the first slot contains only task A. The second slot includes task B in parallel with the sequence of tasks E and F. The last slot contains the sequence of C and G. The total makespan of Scheduling 2 is 460 units of time. Finally, Scheduling 2 has a shorter makespan than Scheduling 1. Considering this example and several others, we could notice the following intuitions:

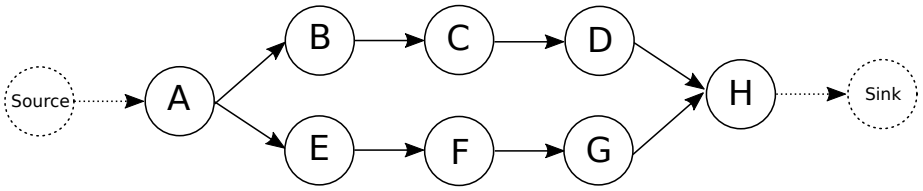


Figure 1.2: 10-tasks DAG example

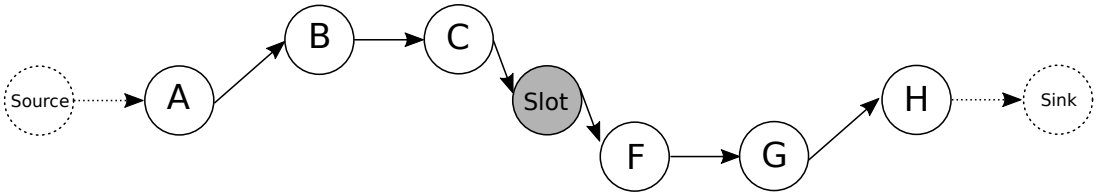


Figure 1.3: DAG of Figure 1.2 after merging task D and task E in slot $Slot$

1. The best solution is not always the one with the lowest number of reconfigurations. Indeed, Scheduling 1 reconfigures the FPGA twice, whereas in Scheduling 2 the FPGA is reconfigured three times.

2. The key metric is the amount of time spent executing tasks in parallel, which should be maximized. In order to do so, the most time-consuming tasks should probably be considered first because the potential gains are the largest. Indeed, in the second schedule the sequence of tasks E and F and the execution of task B (the three most time-consuming tasks) perfectly fit together.

Another interesting example is given in Figure 1.2. For simplicity reasons, in this example each task is supposed to occupy exactly 50% of the FPGA. Let us imagine to schedule tasks D and E in the same slot. This choice totally remove the theoretical parallelisms between the two branches (except for tasks D and E). Indeed, following this choice any parallelism between B and C with F and G would become not valid because of the data dependencies, as shown in Figure 1.3.

It could be convenient though. The best choice depends on how much the gain derived from the scheduling of tasks D and E is with respect to the other tasks of the graph. For example, if the execution times of tasks D and E are similar and much higher than the other tasks, the minimum makespan is obtained by exploiting this parallelism, which implies forbidding any other parallelism. Thus, **the choice of which tasks shall be put in parallel shall not consider only the tasks that could fill the slot, but it should also consider all the tasks of the whole graph.**

Obviously, for larger examples, the exhaustive search may induce combinatory explosion. Indeed many parameters, which can be combined, have an impact on

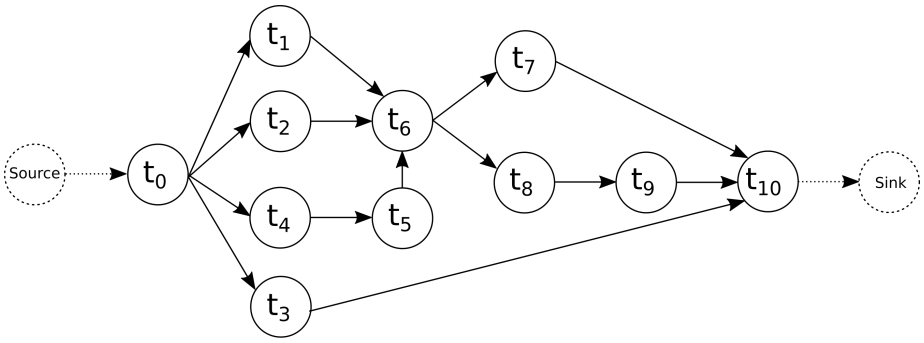


Figure 1.4: A realistic application DAG

Task	LEs	DSPs	EMBs	DRAM	Network bandwidth	Time[u]
t_0	33%	11%	53%	65%	50%	187
t_1	48%	70%	3%	10%	91%	1662
t_i	50%	15%	90%	6%	17%	974
...	...	...	...	...	...	...
t_{10}	22%	78%	10%	49%	9%	563

Table 1.2: Example of resources types and resources distribution in the realistic DAG of Figure 1.4

the final solution. These include data dependencies, parallelism, HETs, requested resources and the availability of resources, reconfiguration overhead, number of reconfigurations and so on.

The objective of our thesis is to define a heuristic that targets specifically the static scheduling of tasks onto an FPGA. We expect our solution to provide a scheduling which is as good as the one that can be computed by meta-heuristics with a runtime which is much lower. Additionally, we also expect to handle situations where the standard deviation between the different parameters of tasks are much higher than the one presented in Figure 1.1 in which for instance all occupancies were similar. An example is illustrated in Figure 1.4 and Table 1.2. Finally, having generic models to represent resources and constraints make the approach more adaptable to various applications and FPGAs.

1.5 Overview of the solution

Our solution is based on an iterative process summarised in the schema in Figure 1.5. Our solution tries to minimize the makespan of an application by grouping tasks in slots. As shown in previous sections, the parallelisms between tasks and dom-

inating tasks (i.e., tasks that have the higher HET) are key-points of the problems that are tackled by our solution. Indeed, our solution relies on the selection of the dominating tasks and it also relies on putting in parallel to these dominating tasks a subset of the tasks than can be executed in parallel.

In order to select which tasks to be put in parallel to the dominating task, we consider all the graph, and not only the quality of the parallelism between the dominating tasks and the tasks in parallel to it. For instance, our decision cosiders the requested resources and HET of tasks, data-dependencies among them, FPGA features and reconfiguration time.

Among all the possible tasks that can be put in parallel with the dominating task, we define in this work a score-based approach that helps defining which tasks to select. Once the dominant task and the tasks to be put in parallel have been selected thanks to the scoring facility (i.e., a slot is created), the graph is reworked so as to merge in a single node all the tasks that have been selected in this iteration.

At each iteration a new slot is created and its tasks are merged together in one node of the graph. The input graph finally becomes a sequence of slots. Such sequence expresses the selected scheduling. As a reminder, a slot contain a total FPGA reconfiguration followed by the execution of tasks belonging to the slot. As shown in Section 4.6.4, a final optimization process tries to further compact the computed slots in order to further reduce the makespan.

Our approach is very likely to identify a good solution, which is the solution that minimizes the makespan, because instead of considering only a given sub-part of the graph, just like when starting from the source task, we rather favour the dominating task (regardless of its location in the graph) and the most powerful parallelism of the graph to better exploiting the parallelism capacity of FPGAs. This heuristic is presented in Chapter 4, while the efficiency of this heuristic is presented in Chapter 5.

1.6 Outline of the thesis

The rest of this thesis is organized as follows. Chapter 2 illustrates the main context of this thesis by providing an overview of reconfigurable computing and how cloud data centers may integrate FPGAs. Chapter 3 discusses the related work in this domain. Chapter 4 describes the main steps of our approach, *Slot*. Chapter 5 experimentally evaluates *Slot* and compares it with other, exact or approximate, approaches. Chapter 6 shows how *Slot* might also be interesting for the design of complex embedded system. In this respect, we have integrated *Slot* in an open-source Model-Driven Engineering tool. Chapter 7 concludes the thesis and it illustrates the

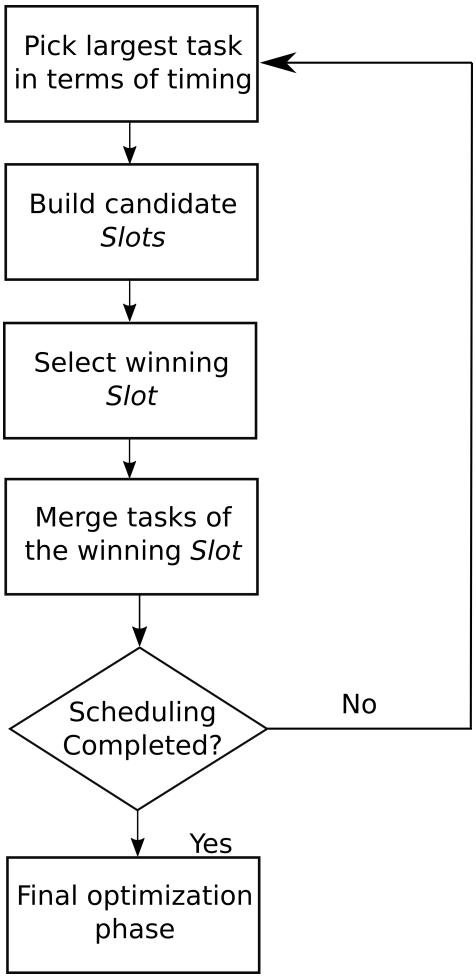


Figure 1.5: The main steps of *Slot*

future work. Finally, Appendix A provides low level details of the MILP formulation that we used to calculate the optimal solution for each test-case of our benchmark and Appendix B contains a list of acronyms used in this thesis.

1.7 Conclusion

As presented in this introduction, cloud data centers may now integrate hardware accelerators such as FPGAs. In ordert to efficiently use these FPGAs, we have also described the problem of scheduling dependent tasks onto these FPGAs. We have assumed that they are totally reconfigured. Unfortunately, as it will be more detailed in Chapter 3, there are not yet good solutions to tackle this problem. Solutions are either precise but far too long to be computed or they are very fast to compute but

they offer solutions in average too far from the optimum. We think that there is an intermediate approach that could compute solutions in a fast way while providing solutions much closer to the optimum.

Then, we have presented the main challenges that such scheduling problems introduce. For this, we have presented several examples. In particular, we have shown that only an algorithm which works on the global graph, that is by taking decision considering all the tasks of the graph together, their resources requirements, HETs, dependencies among them, the features of the targeted FPGA and the reconfiguration time, is more likely to compute a solution close to the optimum, because it can better exploit the parallelism of FPGAs.

We have also in this Chapter sketched our solution, named *Slot*, which is based on an iterative process that first considers all the graph and the dominating task of this graph. *Slot* also relies on a scoring approach to select the best slots among candidates. Also, our approach contains a final optimization stage.

The manuscript first elaborates on which elements of FPGAs must be taken into account for obtaining a good result. We then present in detail the heuristic and we compare it with other, exact or approximate, approaches.

Chapter 2

Context

This chapter illustrates the main context of this thesis. Firstly, Section 2.1 explains the increasing role of hardware accelerators and particularly of reconfigurable computing to better supply the performance requirements of applications. Secondly, Section 2.2 describes the main features of FPGAs, namely the hardware accelerators in which this thesis is focused. Finally, Section 2.3 describes how modern cloud data centers may integrate FPGAs. Also, we describe three real cloud architectures.

2.1 Main Context

Modern computer systems are miniaturizing while being multicore. The current trend is not to use a single and monolithic core, but rather several cores of different nature. The direction is the specialization of each computational unit and the execution of a given task on the most suitable core. This distributed architecture has many advantages in terms of efficiency but new challenges have to be tackled. With the rise of System-on-Chips (SoCs), system design concerns the implementation of several tasks in a single system. Thus, such systems must evolve to better meet the run-time needs of the applications with the combined execution of heterogeneous computational units. In this regard, the *90/10* optimization rule states that *90% of the execution time of a program is spent while executing 10% of the program itself*. In this way, architectures provide designers with the means to speed up the execution of the critical path (intended as the heaviest part of a program from a timing complexity point of view), while maintaining a good flexibility level. Reconfigurable logics arise in this context. A reconfigurable logic is a particular type of hardware device whose circuits can be reconfigured to implement a custom behavior after the manufacturing. The reconfiguration process usually consists in writing configuration information in a particular memory. Reconfigurable computing is joining two different and completely separated worlds: hardware and software. Their

separation is mitigated by the fact that also hardware can be programmable with the reconfigurable computing. The gap between hardware and software is covered by potentially achieving a better level of performance than software while keeping the high flexibility offered by a General-Purpose Processor (GPP). Figure 2.1 positions reconfigurable hardware, in particular FPGAs, between Application Specific Integrated Circuit (ASICs, namely integrated circuits whose function is implemented at manufacturing and never changes) and GPPs from a design time and a performance point of view. Figure 2.2 compares ASICs and FPGAs cost. The number of produced devices is shown in abscissa. Y-axis shows the total cost. $X = 0$ shows *non-recurring engineering* (NRE) costs, that is to say one-time costs for researching, designing, developing and testing a certain product. NRE costs are almost zero for FPGAs (and GPP). This is due to their flexibility. The same FPGA device can be indeed used in principle in order to realize any implementation. The total cost for FPGAs is only given by the cost for their integrated circuit. This is not true for ASICs though. ASICs devices are realized to execute tasks that are known during the manufacturing. In this way they must be designed, studied and tested ad-hoc to meet customer needs. Most of the NRE difference is the licensing of IPs (e.g. ARM cores, I/O controllers...) and the manufacturing of the masks. For circuits using many IP blocks and manufactured in an advanced technology (e.g. 7 nm), all-in-all, it represents several million euros. For ASICs this cost is entirely supported by one company while for FPGAs it is shared among all user companies. NRE costs are therefore very high for ASICs, but slope is flatter. For this reason, integrated circuit of ASICs are more convenient in terms of cost for high production volume. We can conclude that reconfigurable computing is a trade-off between general-purpose and application-specific computation in terms of performance, energy, flexibility and design time.

Reconfigurable computing has enhanced performance of the applications coming from a large range of domains: scientific computing and biological computing, Artificial Intelligence (AI), signal processing among others. The presence of experts in hardware/software co-design is not guaranteed at all in these domains. This means that the future trend will be the definition of a friendly reconfigurable eco-system that opens these technologies to everyone who may take advantage from their use. The contribution of this thesis can also be part of this eco-system. Reconfigurable has become increasingly popular thanks to recent progress in FPGAs development. To sum up, FPGAs are a particular sort of integrated circuits that allow custom hardware implementations. They can be reconfigured a several number of times. Reconfiguration means that they can change their implemented functionality to support a new application. This is logically equivalent to having a new hardware mapped to the chip with a different behaviour. FPGAs allow to have custom-designed high-density hardware within an electronic circuit, with the peculiarity of making its variation possible, even while an application is still running over it. This flexibility and reconfigurability

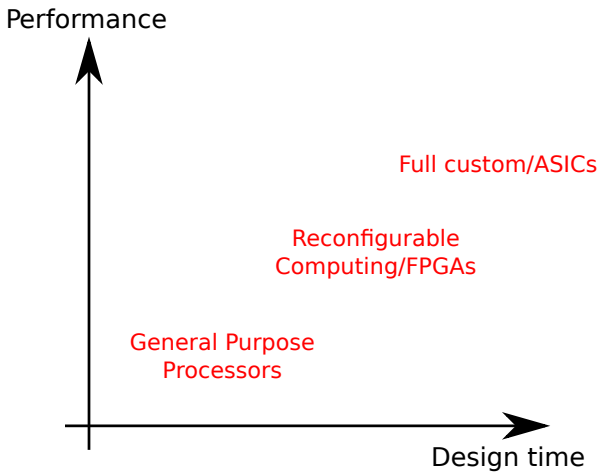


Figure 2.1: Reconfigurable computing as a trade-off between GPPs and ASICs

come at a price. Indeed in the past large implementations in terms of resources (corresponding to large or complex algorithms) were not possible [37]. The lack of a standard programming and architectural model is the main disadvantage nowadays. The absence of high-level programming tools, APIs, standard platforms constitutes the largest obstacle to the extensive use of FPGAs, especially in comparison to other acceleration technologies such as GPUs [105]. With respect to GPUs that are characterized by an easy to use programming model, FPGAs allow higher performance-per-watt and improved hardware acceleration performance.

2.2 A Short introduction to FPGAs

An FPGA is a semiconductor device characterized by programmable logic elements and programmable interconnections. Unlike GPPs, there are neither run-time instructions fetching nor a Program Counter (except by mapping a GPP in the FPGA, such as [10], [9] and [66]). FPGA are programmed to implement required functionalities. In this respect, Intellectual Property (IP) blocks can be used to save design time by using an already existing design. FPGAs are the current state of the art for Programmable Logic Devices (PLDs). A PLD is generically an integrated circuit that contains logic elements of different nature. The latter can be configured and connected each other in various ways. Manufacturing mask sets do not depend on

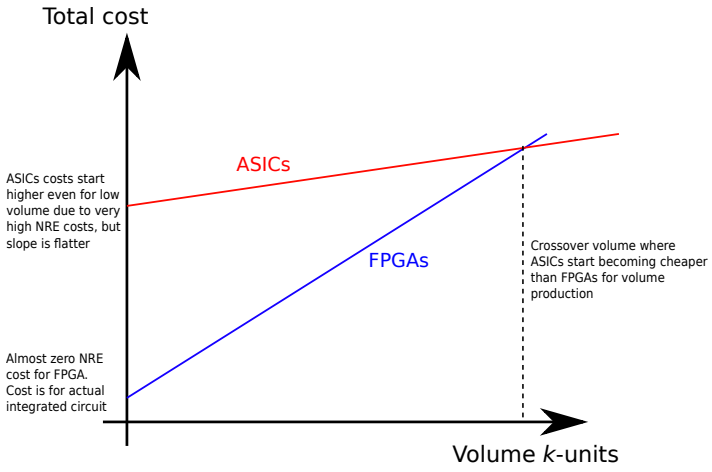


Figure 2.2: ASICs vs FPGAs - production costs and NRE costs

specific end users applications, which are not known during the manufacturing process of a PLD. The same PLD can be used to implement very different functions. The functions are described by end users using the same kind of hardware description languages used for the logic part of ASIC design but end users are almost completely relieved of the heavy physical design tasks.

FPGAs join PLD idea to gate arrays. FPGAs components are grouped in logic elements (LEs). The latter are named in different manner by the most known producers. For example, Configurable Logic Blocks (CLBs) by Xilinx [15] or Adaptive Logic Module (ALM) by Intel-Altera [1].

Programmable interconnections link LEs together and, together with I/O blocks, they represent the main components of an FPGA:

LEs: LEs are the heart of an FPGA. These elements are configured to implement application logic. They can be used to implement both combinatory and sequential logic. A LE is an element that can be customised according to the designer needs. Their combination allows the implementation of complex functions. With regard to Xilinx, a single LE is a hierarchical structure composed of a set of slices. The number of slices forming a LE changes according to the device: they can be two such as 7-Series [18] or one, such as the most recent Xilinx UltraScale+ FPGAs [17]. Figure 2.3 shows the internal composition of a single slice. It contains at least a memory known as Look-Up Table (LUT), a flip-flop, a multiplexer and the necessary interconnection hardware. LUTs are little memories used to implement small combinatory functions. Number of inputs is limited (generally 4 or 6 bits per 1 or 2 bits of output) and it is a trade-off involving, among others, the total size and the actual usability by the synthesis tools for the average design. LUT in Figure 2.3 implement a 4-input,

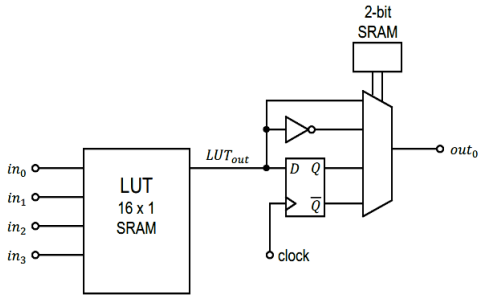


Figure 2.3: Composition of a Logic Element

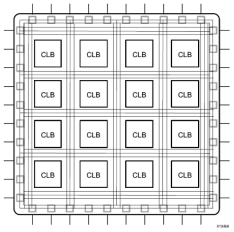


Figure 2.4: LEs organization

1-output function. It is composed of 16 memory cells, typically SRAM, which are written during the configuration phase. One particular input for the LUT of Figure 2.3 that is 4-bits wide basically corresponds to a number in the [0 ... 15] range. On the other hand, a configuration corresponds to what it is stored in the LUT. For a 16 bits LUT, that is a number in the [0 ... 65535] range. It is possible to store 1-bit output corresponding to each specific input regardless of the complexity of the desired function. The presence of a flip-flop in Figure 2.3 enables sequential circuits, whereas the interconnecting logic and gates allows to route signals from and to the LUT. The multiplexer is used to select the output value and the storage of its selector that can be realized with a 2-bits SRAM (namely the same technology which is used for the LUT).

Matrix groups of LEs are finally the backbone of the FPGA, as shown in Figure 2.4[1].

Input/Output blocks: input/output blocks (I/Os) are responsible for connecting signals of the internal logic to a pin which is located in the package of the FPGA. There is only one I/O block for each I/O pin of FPGA chip. I/O blocks are usually also configurable in direction, voltage, signaling standard, etc. In order to support

[1] LEs are named using the Xilinx term Configurable Logic Block - CLB

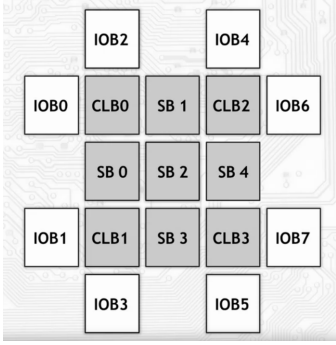


Figure 2.5: FPGA interconnections - Switch boxes

this the I/O blocks contain buffers, multiplexers, etc. which configuration is stored in SRAM-like memory elements. Figure 2.7 illustrates the position of such I/O blocks with respect to the rest of FPGA architecture.

Interconnections: programmable interconnections allow arbitrary connections among LEs and between LEs and I/O blocks. There are two main interconnection models: directed and segmented. Directed interconnections cross the device in all dimensions. Vertical and horizontal links among LEs shown in Figure 2.4 are an example of this. Once a computation has finished, LEs inject data onto the most suitable channel for a certain destination (e.g., an I/O block or another LE). This implementation also includes additional short channels connecting neighbouring blocks. On the other hand, segmented interconnections are composed of lines that interconnect through programmable switch boxes, as shown in Figure 2.5. Switch boxes are implemented using pass-transistors that can be enabled or not, depending on the configuration. Directed interconnections are characterized by amost constant parasite resistance and parasite capacity and that leads to better predictability of signal propagation time. On the other hand, the segmented model is characterized by less power dissipation. Indeed, resistance and capacity of interconnection lines are only those of interconnections between boxes. The limited leght of such lines is due to the distribution of several boxes onto FPGA area. A mixture of both models is the most common trend nowadays [98].

Modern FPGAs are not just an array of interconnected reconfigurable elements and I/O blocks. They also frequently contain additional resources. As shown in Figure 2.7, in addition to the logic elements, an FPGA can embed larger dedicated hardware blocks. These can be RAM blocks, optimized integer or floating point arithmetic units (frequently referred to as "DSP blocks"), clock synthesizers, etc. Some FPGAs even embed complete GPPs. Note that these FPGAs with a GPP as a hardware block (hard core) are not the same as an FPGA in which the reconfigurable logic is used to implement a GPP (soft core). The performance of a hard core is

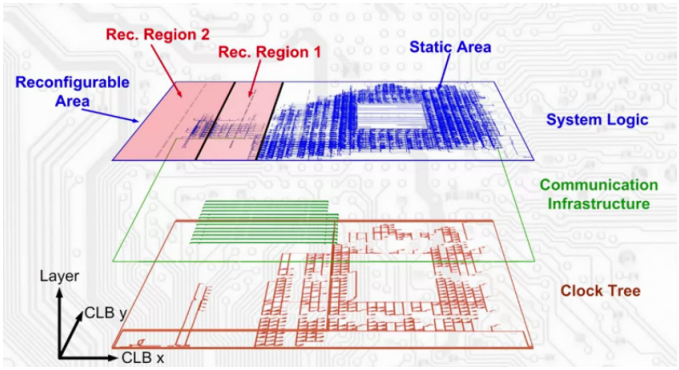


Figure 2.6: FPGAs architecture - a global view

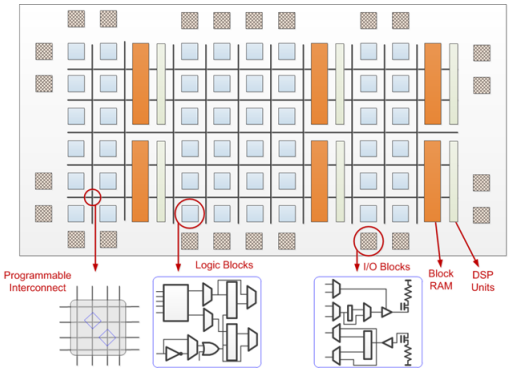


Figure 2.7: Main components of FPGAs architecture

much better, it is smaller and consumes less power than the equivalent soft core. In this way, system designers can take advantages from the hardware already present in the circuit and can integrate it in their designs without implementing all the desired functions onto the reconfigurable part of FPGA. Hard-cores of an FPGA enrich its functionalities and they enhance both the speed of the architecture and the design speed. It is important to add two things to complete this short FPGA introduction: how FPGAs are physically configured and how system designer can describe the desired behaviour of circuits. Memory cells, which are adjacent to LEs in Figure 2.7, drive their key-features. LEs behaviour such as equations of LUTs, I/O blocks and interconnections are driven by particular bit values stored in memory cells (known as configuration memory). Configuration memory is composed of SRAM blocks. Thus, it is a volatile memory. Upon a device restarting, all its configuration is lost and a new one must be downloaded onto the configuration memory. The most common trend provides an external non-volatile storage from which the FPGA downloads its configuration through a dedicated interface. Once the download is complete, the FPGA starts executing the application defined by the current configuration.

FPGA configuration, that is the information copied onto the SRAM configuration memory, consists of a file known as *bitstream*. A bitstream file can be either *full* or *partial* depending on the scope of the reconfiguration taken into account. A full bitstream reconfigures the whole FPGA at once, while a partial bitstream reconfigures only portions of the FPGA. When the reconfiguration can be done during the execution of an application, we say that the FPGA can be dynamically reconfigured. As shown in Figure 2.6, the reconfigurable part of an FPGA can be divided into independently reconfigurable zones known as regions. Different regions can be characterized by different frequency, depending on the clock infrastructure of the FPGA. A region bitstream is the sequence of bit used to configure a reconfigurable region. Of course designers do not write configuration bistreams by hand. They use Hardware Description Languages (HDL), which are high-level languages dedicated to the description of the structure and the behaviour of electronic circuits. The descriptions written in HDL are simulated to verify that they really model the expected behaviour and, finally, they are translated into a network of interconnected logic elements (a "netlist") by a software tool called a logic synthesizer. In ASIC design flows these netlists are the entry point of the physical design that leads to the production of the physical masks used to create integrated circuits. In FPGA design flows they are the entry point of a shorter and simpler physical design where the netlist elements are mapped to FPGA resources and interconnected using the reconfigurable interconnect. HDL languages have many similarities with high-level computer languages but there are also important differences like, for instance, the parallelism or the time which are fundamental aspects and at the heart of HDLs while they are frequently provided by external libraries or calls to the operating system in traditional programming languages. The most common HDL languages for ASIC and FPGA design flows are VHDL [21] and SystemVerilog [20].

2.3 FPGAs in the Cloud

Modern FPGAs are not merely a container of reconfigurable hardware but they can also embed other elements such as general-purpose processors, DSPs, and so on. One among the pioneers of this heterogeneous software-reconfigurable hardware nature is Xilinx Virtex II Pro device, where FPGA includes an IBM PowerPC405. Several improvements have been done since. For instance, Stellartron products distribute an Intel Atom E6XX processor with an FPGA Intel in the same package [59]. The last trend for high-performance computation is represented by projects such as Cygnus [35], a supercomputer that has been developed at Center for Computational Sciences (CCS), in Tsukuba, which embeds a mixture of CPUs, GPUs and FPGAs. Supercomputing is not the only domain in which processors and accelerators works together, because this is very common in cloud infrastructures. Here FPGAs are

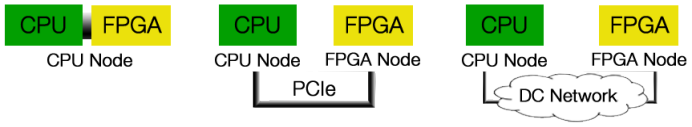


Figure 2.8: Three possible connection scenarios between CPUs and FPGAs in cloud data centers

essentially coupled with one or more host CPUs via a fast connection such as PCI Express (PCIe). This scenario is transforming FPGAs from devices which are only used within the context of a single System-on-Chip, (e.g., left-hand side of Figure 2.8) into a custom accelerator for the code running on the CPU as shown in the middle picture of Figure 2.8. In both cases the FPGA can be used as a custom accelerator for the application running on the CPU. The difference is that the CPU(s) can be or not in the same integrated circuit as the FPGA(s). With the configuration in the middle picture of Figure 2.8, FPGA resources are ready to be used upon deployment and can be elastically scaled. Thanks to their ultra-large scale resource pool, they can meet the needs of greater numbers of FPGA resources whether necessary (e.g., it is possible to offload a part of the computation to a remote FPGA, as in [33]). The reconfigurable nature of FPGAs can finally provide a suitable fit during changing of workloads in modern data centers. Despite all these advantages, FPGAs have been only recently introduced into cloud infrastructures. Microsoft was indeed one of the first *large companies* to do it in 2010. The vast majority of other important companies (e.g, Amazon and the three major Chinese companies Alibaba, Tencent and Baidu) followed this trend in the late 2016 and in the early 2017, as can be seen in Figure 2.9. This was due to the costs of FPGA engineering. FPGA design and development was definitely not an easy task few years after the dawn of this technology, and they could host relatively small designs [37]. In the last decade, FPGAs are rather used as computational elements within cloud scenarios or complex heterogeneous systems which include general-purpose processors too [36].

The coupling of one or more FPGAs with one or more processors has become the basic node of a distributed architecture. As can be seen in Figure 2.9, several companies have shown interest in these distributed and heterogeneous architectures. In the rest of this section, we will briefly explore the basic node of two of them: Microsoft Catapult and F1 instance of Amazon Elastic Compute Cloud. We will focus on the architecture of the node itself and on how nodes are connected in the distributed architecture. We have chosen these particular architectures because they are the closest to those targeted by this thesis. Although they are similar in principle, they are used by companies with different targets and technologies.

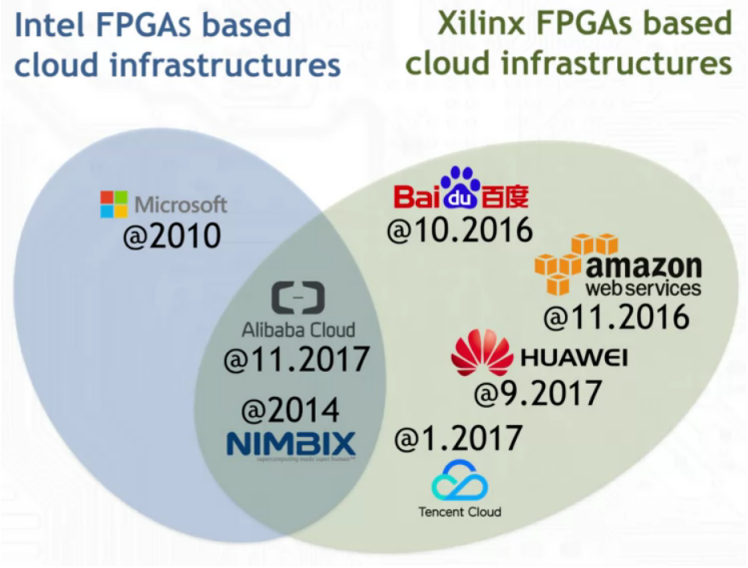


Figure 2.9: The integration of FPGAs in some cloud data centers

2.3.1 Microsoft Catapult

FPGAs are being deployed in Microsoft's data centers and the idea is the acceleration of Bing, Azure, processing and network speeds [92] [33]. Instead of using potentially less efficient software to implement intermediate services, using FPGAs Microsoft engineers can implement their algorithms directly onto the hardware. By doing this with a programmable device, instead of using an ASIC, they can gain advantages by the device reconfiguration. Figure 2.10 shows the architecture of a basic node of the distributed architecture and how it is connected to the rest of it. A single server includes two CPUs, each one with its own memory. FPGA has three type of connections, one towards the host CPU, one with the Network Interface Card (NIC) of the server and one with the rest of the data center (Top-Of-the-Rack - TOR network switch). The FPGA is connected to the host CPU through a PCIe connection and this allows the CPU to exploit the FPGA as a local accelerator. Connections with the NIC and with the TOR network switch allows the FPGA to be tightly coupled with the network of the data center. In this way, servers that do not exploit all the FPGAs resources can make the rest of them available to remote servers. Microsoft engineers has shown how remote FPGAs can be accessed with a latency in the order of few microseconds. The target FPGA is an Intel (Altera) Stratix V D5, characterized by 172.6K of LEs (named Adaptive Logic Modules - ALMs, using Intel convention), 4770 DSPs, one channel to a 4 GB DDR3-1600 DRAM, two PCIe links, which supports a bi-directional bandwidth of up to 16 Gb/s between CPU and FPGa and two 40 Gb Ethernet interfaces towards th NIC and the TOR switch [2]. Microsoft designers

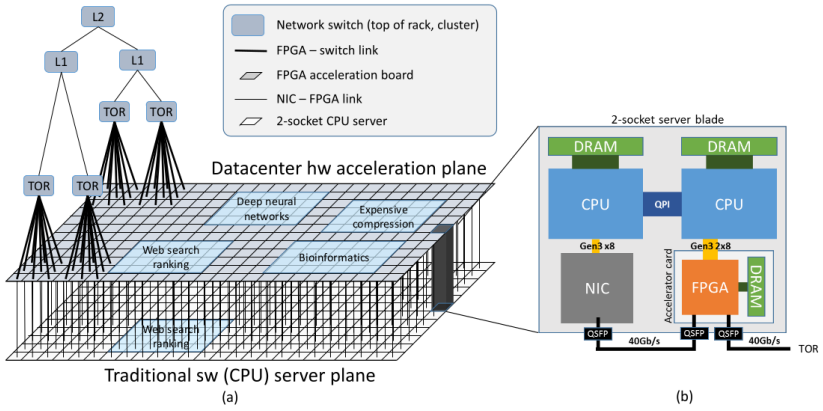


Fig. 1. (a) Decoupled Programmable Hardware Plane, (b) Server + FPGA schematic.

Figure 2.10: Microsoft Catapul cloud data center architecture, from [33]

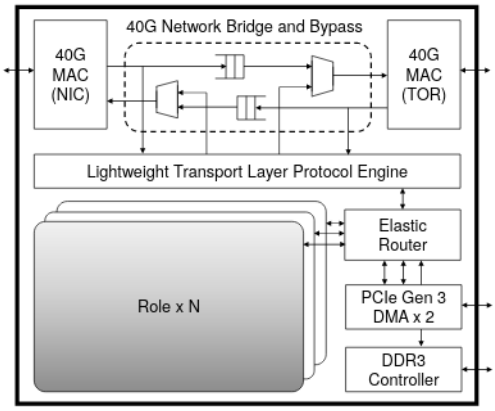


Figure 2.11: Division Shell-Roles in Microsoft Catapul FPGAs, from [33]

have logically divided the FPGA into a static part (named *Shell*) and several reconfigurable parts (known as *Roles*). Roles host the application logic, whereas the Shell implements all the logic used by the services that are going to be accelerated. In this respect, Roles cover around 32% of the LEs of the FPGA and they are the only ones being reconfigured. The rest is covered to implement the Shell, which includes the memory controller (around 8%), the communication interfaces, and so on. Shell uses reconfigurable resources to implement the supporting logic but there is no need to reconfigure it in order to change the application. Figure 2.11 shows the division between Shell and Roles.

2.3.2 Amazon Elastic Compute Cloud (F1 instances)

In this part we deepen the architecture of Amazon Elastic Compute Cloud [4]. We will specifically focus on the basic node of the distributed architecture, namely F1 instances. In contrast with Catapult Project where FPGAs are used by Microsoft to accelerate Bing ranking or network services, FPGAs parts of F1 instances can be directly accessed by end-users who can implement their custom designs. Here FPGAs are available to users that want to customise and rent them as a cloud service. This implements a sort of AaaS (Acceleration as a Service). Briefly, designers must create an FPGA design through a special virtual device known as Amazon Machine Image (AMI). Then, the user must register her design as an Amazon FPGA Image (AFI), and he can finally deploy it to the F1 instance, the heart of the acceleration. Once an Amazon FPGA image has been developed, this can be made available to other customers on the Amazon Web Services (AWS). The acceleration is accessible to software developers with little to no FPGA experience, thanks of the Xilinx High Level Synthesis design flows that allow designers to use OpenCL, C and C++ as their design entry languages to accelerate their applications onto Amazon F1 instances [22].
Figure 2.12 shows the architecture of a F1 instance and its connection with other nodes of the data center. F1 instances can contain one, two or eight FPGAs (the latter is the case illustrated in Figure 2.12). Each FPGA is connected to a 8-cores vCPU over a PCIe link. The FPGA chosen for this architecture is a Xilinx Virtex Ultrascale 9P [16]. AA single Xilinx Virtex Ultrascale+ 9P FPGA provides about 150,000 Logic Elements (LEs), that is, about 1.2 millions 6-inputs LUTs and 2.4 millions Flip-Flops, more than 6800 DSP slices and 75.9 Mb of total block RAM. The whole F1 instance provides up to 122/976 Gb of DDR memory plus up to 470/4*940 Gb of SSD, according to the number of FPGAs in the particular F1 instance. Each channel towards the DDR memory is characterized by a total bandwidth of 48 gigabytes per second.

It is interesting to notice how Amazon has affected the research of other realities. It is the case of Ryft with the Ryft One project [12], a Big Data infrastructure obtained via a Xilinx FPGA-accelerated architecture. Ryft has developed Ryft Cloud, an accelerator for data analytics and machine learning that extends Elastic. Ryft Cloud sources data from Amazon Web Services (such as Amazon Kinesis [5], Amazon Simple Storage Service (s3) [6], Amazon Elastic Block Store [3]) and uses massive bitwise parallelism to drive performance.

2.3.3 Future trend in distributed systems using FPGAs

In most of these servers architectures each FPGA is connected to a CPU through a high-speed point-to-point connection (such as PCIe). FPGA is a co-processor slave

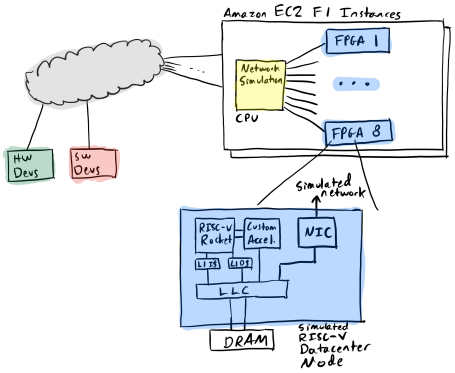


Figure 2.12: Amazon Elastic Cloud 2 - F1 instance, from [4]

under the control of a master CPU. This situation is shown in the middle of Figure 2.8. Because of this master-slave paradigm, such FPGA is integrated in the cloud as a computational option of its owner device. As a matter of fact, FPGA can only be accessed through an indirect manner such as containers or virtual machines (VMS), e.g., F1 instances of Amazon Elastic Cloud. New research trends are overturning this paradigm. An example is the cloudFPGA initiative of IBM [23] [97]. Right-hand side of Figure 2.8 shows the idea behind cloudFPGA. Here FPGAs are independent devices attached to the network. So, they can be directly accessed by end users similarly to CPU VMs or Containers. The architecture of Microsoft Catapult presented in Section 2.3.1 actually goes in this direction by connecting each FPGA with the network of data center. Each FPGA remains connected to a master CPU though. "cloudFPGA" initiative completely disaggregates the FPGA from the CPU. FPGA has then become a completely standalone computing unit. In this regard, there is no more need of a power-hungry CPU associated to each FPGA. This leads to a rise of distributed FPGA devices. Figure 2.3.3 shows the architecture of cloudFPGA. We can identify, in green, three main actors, at different layers of granularity: a Data Center Resource Manager (DCRM), a Sled Manager (SM), and an FPGA Manager Core (FMC). There is only one DCRM in each data center. It stores user images, it knows FPGA resources and it drives several SM. Each SM drives in turn a Sled, which is a group of 32 FPGAs. It is responsible both for the power cycle and monitoring of each device. Each FMC collaborates with the other two layers to provide demanding tasks with the requested FPGA resources. FMC contains a simple HTTP server and it communicates with the rest of infrastructure through a set of RESTful APIs. REST has been chosen because it is a platform independent framework proven to scale well with large volumes (e.g., CPU-based applications in the Web).

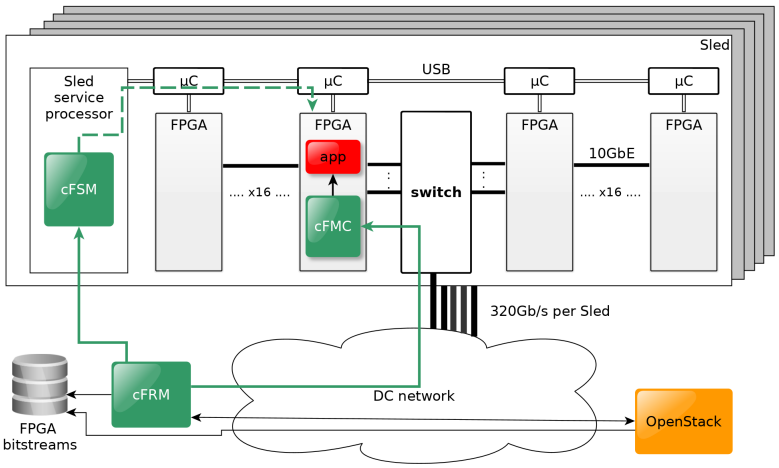


Figure 2.13: System architecture of the IBM *cloudFPGA* platform, from [97]

Chapter 3

Related Work

3.1 Introduction

The scheduling problem we target in this thesis is similar to Resource-Constrained Scheduling Problem (RCSP) category. RCSP problems are scheduling problems whose scheduling is influenced by the availability or lack of resources. Often this means that due to resource limitations a certain application will take longer. Section 3.1.1 generally define the RCSP and it introduces how our specific FPGA scheduling problem fits onto this classification.

The rest of related work is composed of two different aspects. Section 3.2 analyzes how each entity (i.e., application and architecture) is modelled within contributions on FPGA scheduling. In other words, Section 3.2 presents an overview about inputs, outputs, assumptions, models and parameters which are necessary to calculate a scheduling of applications onto FPGAs. Models are used to describe both the applications and the architecture. With regard to applications, we focused on presenting modelling assets which allow to capture input parameters that are relevant for solving a scheduling problem. Models used in scheduling problems have generally more coarse-grained parameters than models used by works whose goal is, for instance, generating an hardware implementation FPGA-specific for a specific application. Similarly for architectures: as we will see, FPGAs are characterized by a complex architecture and not all the details are relevant for scheduling. In this respect, Section 3.2 describes how related works models FPGA architectures. Imprecise models (i.e., models that does not consider details which are relevant for the scheduling) could lead to sub-optimal or even erroneous scheduling solutions.

Modelling is important but with respect to scheduling, the algorithm used to solve the scheduling problem is important too. In the following we designate this solving algorithm the "scheduling strategy". Thus, Section 3.3 illustrates main works of the state of the art dealing with scheduling strategies and it represents the largest part

of this chapter. We have classified related works based on the scheduling strategy they use. Overall, a scheduling strategy can be based on **meta-heuristics** (MHs, Section 3.3.2) or on **heuristics** (Section 3.3.1). We further classify the latter in **list-based** heuristics (Section 3.3.1.1) and **packing-based** heuristics (Section 3.3.1.2). Some of these categories are preferred for the high-quality solutions they provide (e.g., meta-heuristics, exact formulations) or for the quick run-time (e.g., list-based scheduling algorithms). The contribution of this thesis can be placed in "packing-based " category, i.e. algorithms that take decisions by considering groups of tasks. This permits to achieve results that are comparable with meta-heuristics in terms of quality while keeping the run-time comparable with list-based heuristics. Not to be incomplete, exact mathematical formulations can be used as the problem specification by non-exact heuristics too.

Several related works are based on assumptions which are different from those taken in this thesis, but still relevant for our research work. These assumptions help in categorizing similar works. In this regard, many of them target partial reconfiguration ([96], [43], [93], [55], [58], [82], [27], [102]). Partial reconfiguration allows a limited, predefined portion of an FPGA to be reconfigured while the configuration of the remainder of the device is unmodified. Note that this is different from the concept of dynamic configuration in which the device is reconfigured during operation. Partial reconfiguration provides an advantage over multiple full bitstreams in applications that require continuous operation of the FPGA, which would not be possible during full reconfiguration. One example of this could be the baseband processor of a 5G receiver mapped in an FPGA and in which the channel decoder block is dynamically reconfigured to implement a different decoding algorithm or a different variant of the same decoding algorithm while the other blocks (FFT processor, descrambler, QAM de-mapper, channel estimator...) continue their operations. Taking into account partial reconfiguration is a completely different research problem from ours because there are no more reconfigurations that separates entire groups of tasks: reconfigurations of a portions of the device may occur in parallel to tasks execution and their overhead can be partially hidden by loading a task (namely configure an FPGA portion to execute a certain task) before the time in which such task is effectively ready to start its execution (reconfiguration prefetching). If characteristics (e.g., timing, resources, dependencies and so on) of tasks that compose the application are known a priori, the space of solutions is even larger than the corresponding with full reconfigurable FPGAs.

Knowing tasks features a priori is a common situation in cloud computing contexts. According to [112], there are families of cloud applications that launch a simultaneous and large number of dependent tasks on an accelerator or they aggregate batch of tasks to accelerate such batch. This implies that features of such batch are surely known before their acceleration. However, some contributions ([55], [58]) do not consider an application whose tasks and tasks characteristics are known a

priori, as assumed in this thesis, but rather they schedule tasks by reconfiguring a subset of the FPGA resources according to their requirements in terms of resources, which are known only at the time of arrival. In contrast to this, we will address the scheduling of application by assuming we have a complete knowledge of resources and dependencies of tasks which are assigned to an FPGA at a given moment.

The remaining criteria which contribute to catalogue the related work are: computational complexity of models, presence of data dependencies among tasks and the scheduling goal. About the latter, this chapter includes contributions whose target is not makespan minimization but similar in the sense that they consider timing objectives. For instance, common goals of other contributions are the minimization of the reconfiguration numbers, the respect of deadlines, the reduction of hardware waste for each reconfiguration stage, the minimization of rejection rate in case of online scheduling and so on.

In this section we have first chosen to divide the different approaches according to the approach to solve the scheduling problem (i.e., exact formulations, metaheuristics, heuristics). Within each category, further classifications are made according to the scheduling goal (e.g., makespan minimization), to the nature of the reconfiguration (i.e., full or partial reconfiguration) and to on-line/off-line scheduling of applications (refer to Section 1.3). Table 3.1 gives a global overview of the family to which every work belongs according to the previously introduced classification.

Section 3.4 finally concludes this chapter. A resume of existing approaches is provided along with a discussion of their unsuitability for the FPGA scheduling problem targeted in this thesis. We chose to not dedicate a section to them because exact solutions are radically different and do not really compare with the other works. However, as we will see in Chapter 4, we will use an exact strategy to compute the optimum solutions and serve as a quality reference in our benchmarks.

3.1.1 Resource-Constrained Scheduling Problem

The scheduling problem we address in this thesis is similar to Resource-Constrained Scheduling Problem (RCSP). As explained in [11], a RCSP is a combinatorial optimization problem that consists in minimizing a schedule for an application composed of dependent tasks, characterized by known parameters in terms of execution time and required resources. Such schedule is subject to constraints given by the limited availability of the resource and some of them must be renewed to execute new tasks (e.g., the reconfigurable area of an FPGA). “More formally, the RCSP can be defined as a combinatorial optimization problem. A combinatorial optimization problem is defined by a solution space X, which is discrete or which can be reduced to a discrete set, and by a subset of feasible solutions $Y \subseteq X$ associated with an objective function $f : Y \longrightarrow R$. A combinatorial optimization problem aims at finding a feasible solution $y \in Y$ such that $f(y)$ is minimized or maximized.” [11]

The RCSP belongs to the NP-hard class of problems [52]. In the general case it means that its decision version (deciding if a schedule with a makespan less than a given bound exists) is NP-complete. More concretely it means that when the size of the problem instances increase (e.g. the number of tasks), finding optimal solutions quickly becomes intractable.

Our FPGA scheduling problem resembles but is not exactly a RCSP problem because of two differences:

- The configuration slots cannot overlap in time: before any task of a slot can start, all tasks of the previous slot must complete. While in the schedule of a RCSP problem it can be that there is always at least one running task.
- The reconfiguration time is not zero and will have to be considered in order to minimize the total makespan.

Because of these differences, and because we did not find a proof of equivalence, we cannot claim that our FPGA scheduling problem has the same complexity as RCSP. However, our experiments with classic exact solving methods (Mixed Integer Linear Programming) show that the run-time of the solver increases very rapidly with the size of the problem instances. This is a good indication that the problem is hard, that efficient heuristics are needed and that the trade-off between the heuristics' run-time and the quality of their results will be a key aspect. In order to propose solutions to the FPGA scheduling problem we need a model, which abstracts the application and the underlying architecture, and a scheduling strategy. Related work on modelling is presented in next section, namely Section 3.2, whereas scheduling strategies are presented in Section 3.3.

3.2 Modelling

Digital design for FPGA targets can take many forms and use different Models of Computation (MoC) and different types of tools, from logic synthesis of Register Transfer Level (RTL) descriptions expressed in HDL, to high-level synthesis of algorithmic descriptions [68]. In all cases the final result is too detailed and complex for the kind of analyzes we are interested in. The use of abstract models allows to solve problems by focusing on parameters that are relevant (e.g., HET and reconfiguration time for our FPGA scheduling problem, among others), without considering low-level implementation details.

Modelling regards both the input applications and the FPGA architecture. For example, authors in [47], [45] propose the analysis of processing graphs applications onto FPGAs by focusing on efficient computational models to capture all the details of a hardware application. Their research problem is finding a computational model

to represent applications in the form of direct acyclic graphs (DAGs) that efficiently capture low level details with respect to a specific FPGA. Through it they aim to verify the correctness of a data-stream output subject to hardware constraints given by a real FPGA device. Indeed, this task is often subject to a large work of debug directly done by developers (e.g., through test-benches) and it is tedious and time-consuming. They analyse computational models of the family of Synchronous Data Flow (SDF) [76], included Cyclo-Static Data Flow (CSDF) [29] and Static dataflow with Access Patterns (SDF-AP) [54]. According to the authors of [47], with respect to efficiency, scheduling strategies play even a more important role than the computational models themselves.

To this respect, the abstractions provided by SDF models can be inadequate and can lead to inefficient scheduling from the makespan point of view. It is common practise associating the worst-case execution time (that corresponds to the time we defined as HET in FPGA scheduling problems) while to the standard SDF when analyzing the timing behavior of tasks which compose an application [108], [90], [87]. Along with such timing information, these timed-SDF models can be used to capture the behavior of hardware (or software) tasks to specific architectures according to resources constraints. However, such timed-SDF models suffer of an important drawback that is the loss of information about the precise timing of consumption/production of tokens (i.e., data exchanged between tasks). Indeed, for hardware implementations, data tokens should be delivered to them at precise clock cycles, such as in [77], but such SDF models can only describe the number of needed and produced tokens. The SDF model assumes that a task (i.e., an actor using SDF terminology) should wait to execute until there are sufficient tokens available at the inputs. For example, if a task B must receive data from a task A. A produces one SDF token per clock cycle and B lasts for x clock cycles and consumes x tokens per firing, $x > 1$. An efficient implementation of such behavior can be achieved by a FIFO queue of size 1, but the timed version of the original SDF cannot capture such behavior: in this example, task B must wait to receive x tokens in x clock cycles before firing. This can lead to inefficient makespan estimations that are longer than reality. Enhancements of such timed-SDF models such as Cyclo-Static Data Flow (CSDF) and Static dataflow with Access Patterns (SDF-AP) try to overcome such limitations by considering the presence of buffers between tasks. However, even simple designs can lead to huge or infinite buffers and that makes the scheduling solution unfeasible. Finally, they propose their own model of computation named Actors with Stretchable Access Patterns (ASAP) [46], an extension of SDF-AP model. The use of Access Patterns permits to capture the exact moment at which input/output tokens are produced. These works ([47], [45], [46]) is particularly interesting and complementary with respect of the research goal addressed by this thesis. Indeed, they focus on finding the most appropriate computational model for applications to capture hardware constraints in order to achieve a correct design and a correct scheduling, whereas

this thesis focuses more on the scheduling strategy.This means that the proposed ASAP model presented in [46] can be used as a input for the scheduling methods presented in this thesis, especially in an extension of our work (ref. to future work in Section 7.2). In turn, scheduling methods presented in this thesis can be used by [47] to generate an efficient scheduling that is assured to be feasible and efficient by their verification methods. Last, [47], [45], [46] focus on the applications only and not on architectures. In our FPGA scheduling problem, some timing parameters are determined by the architecture and therefore they cannot be captured by application modelling. This is the case of the overhead given by reconfigurations, which are in turn dictated by the resources demanded by each task and by the availability of the FPGA, the contention given by concurrent accesses to common resources such as memories, delays introduced by buffers and so on.

Along with the abstractions used to describe an application, abstractions are also used to describe the architecture, in this case an FPGA. Even though there exists modelling languages which are specialized on abstract hardware description, such as MARTE [8], scheduling problems for FPGAs usually require architetural models which are simple and very coarse-grained. For this reason, we discussed of hardware descriptions based on MARTE in Section 6.1, where we show a possible integration of our contribution in Model-Driven Engineering for the early design of embedded systems. As it can be seen in the fifth column of Table 3.1, several works ([41], [96], [93], [64], [62], [63], [42], [55], [58], [51], [112]) in this domain represent the FPGAs with a mono-dimensional model capturing the area of FPGA in terms of number of Logic Elements (LEs). Tasks that are scheduled onto the FPGA requires a certain percentage of FPGA area and this is considered enough to establish how many and which tasks can fit the FPGA area at the same time with no need to reconfigure. These simple models are related to what the FPGAs offered when the models were proposed, as the column *year* in Table 3.1 suggests. However, even more recent contributions (e.g., [112], [96]) considers very simple models. Even though we agree that very fine-grained hardware representations are not necessary, we consider such mono-dimensional model extremely limiting with respect to the reality. However, more recent In this respect, we chose to model an FPGA with a nD-size array of resources. Each resource can represent a scheduling-independent resource such as number of LEs, number of DSP or number of Embedded Memory Blocks (EMBs) available on a certain device. In future, our model will tackle scheduling-dependent resources too. Naturally, because this thesis is focused on makespan optimization, an FPGA is labelled with all the information needed to determine the reconfiguration time such as bitstream size, clock frequency for the reconfiguration interface and size of the data bus towards the reconfiguration memory. Reconfiguration time has a non negligible impact on the final makespan of an application.

3.3 Scheduling

3.3.1 Heuristics

3.3.1.1 List-based scheduling

In list-based scheduling, individual tasks are sorted in a priority list and assigned, in sequence, to the earliest available unit that fits their resource request [26]. Priorities can be assigned statically or dynamically based on different characteristics, e.g., execution time, resource occupancy and subject to data-dependencies among tasks, if they are present. With respect to FPGA scheduling problem, an available unit is a reconfiguration stage that can physically satisfy the resources required by a target task. This process is repeated until a valid scheduling is obtained. List heuristics are very popular, they require very small run-times in exchange for the optimality of the output schedule. Priority list can be calculated in several ways. For instance, order of tasks can be driven by deadlines, if applicable (e.g., Earliest Deadline First - EDF). EDF algorithm, which executes tasks according to their absolute deadline, has proven to be optimal in a single processor real-time system [19]. For a multi-processor system with m machines, EDF, which executes the first m tasks with the m strictest deadlines among the ready tasks, is not optimal though [65]. FPGA scheduling problem looks like to multi-processor scheduling but it differs on several aspects. For instance, FPGA reconfigurations are different from switching penalities in software scheduling. On one hand, switching penalities in software scheduling are not constant and they are difficult to predict. On the other hand, FPGA reconfigurations add an overhead on the overall makespan that can be comparable to the HET of a task. Always in case of heterogeneous workers (e.g., multi-processors or different regions within an FPGA) a well known list scheduling strategy is Heterogeneous Earliest Finish Time (HEFT) algorithm. Our own classification of list-based scheduling heuristics is:

- **Timing-based:** tasks are ordered according to their data dependencies and their execution time. For example, parallel tasks can be ordered by increasing or decreasing hardware execution time.
- **Resource-based:** similarly to *timing-based* approach, tasks are ordered according to their data dependencies and their resource consumption. It is possible to privilege largest tasks or shortest tasks, for instance.
- **Critical resource-based:** *resource-based* approach has a global nature that statically considers resource utilization. Ordering can be done by considering the availability of the different resources during the construction of the scheduling.

Tasks within the priority list have to be interleaved by device reconfigurations in order to make the scheduling feasible. Starting from the beginning of the list, consecutive

tasks in this list form a reconfiguration stage. When there are not enough resources for the next task in the list, device is reconfigured and another reconfiguration stage is instantiated. Ordering tasks according to their data dependencies allows to build a feasible scheduling. Parameters such as timing, resources, and so on can be combined in more complex ways than the one presented so far. However, regardless the strategy used, list-based strategies have a common flaw that make them strongly sub-optimal for the FPGA scheduling problem addressed in this thesis. Using a list-based strategy to address our FPGA scheduling problem, which targets the makespan minimization onto fully reconfigured devices, requires to first create a priority list and second to interleave total reconfigurations when resource requirements of consecutive tasks cannot be satisfied by the FPGA device. The problem is that in such a way the total reconfiguration is driven by constraints on resources and therefore it is performed only when necessary, regardless of parallelism among tasks or data dependencies. The consequence is that the device reconfiguration can occur in such a way that separates two parallel tasks whose parallelism is particularly convenient. For these reasons, list-based heuristics are more common in FPGA scheduling problems that target partial reconfiguration: indeed, the risk that a full reconfiguration acts as a *wall* between two tasks is not applicable. However, some works applies list-based scheduling to fully reconfigured devices, such as [41] and [64].

Paper [41] proposes an adaptation of the well known Earliest Deadline First (EDF) algorithm to schedule periodic real-time tasks onto FPGAs with the aim to respect the larger number of deadline possible. Each task is labelled with a hardware execution time HET, a period time P (i.e., the interval of time between successive occurrences of the same task) and the proportion $0/leA \leq 1$ of a unique hardware resource (e.g. LEs) it requires. The authors propose two utilisation metrics for a task set Γ:

1. Time utilization factor: $U_T(\Gamma) = \sum_{T_i \in \Gamma}(HET_i/P_i)$.
2. System utilization factor: $U_S(\Gamma) = \sum_{T_i \in \Gamma}(HET_i * A_i/P_i)$.

In the extreme cas where all tasks are executed sequentially $U_T(\Gamma)$ represents the fraction of time in which the FPGA is used. $U_S(\Gamma)$ represents the time-area resource utilization factor for a sheduling of Γ, with or without parallelization. As explained by the authors, if $U_S(\Gamma) > 1$ there is no feasible schedule. Authors of [41] adapted original EDF algorithm to the *Next-Fit* (NF) version. EDF-NF maintains a list, sorted by absolute increasing deadlines, which contains tasks that are *ready* but still not *running*. EDF-NF scans the *ready* list to select which task to add to the list of *running tasks*: tasks are selected to run as long as the resulting set is feasible, according to $U_T(\Gamma)$ and $U_S(\Gamma)$ utilization metrics. When a task cannot be scheduled, EDF would reconfigure the device and starting again the process. EDF-NF acts differently because, when the following task in the *ready* list cannot be scheduled

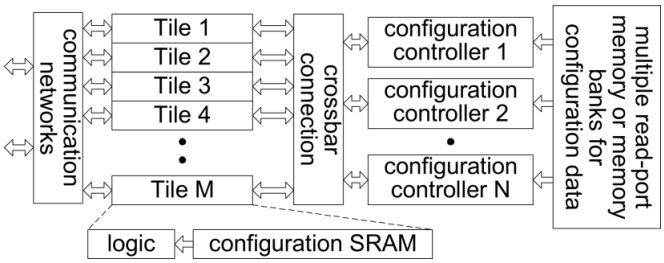


Figure 3.1: Architecture model of [93]

because of constraints on physical resources, it continues to scan the list in order to find other tasks, with an absolute longer deadline, that can be run with the current list of *running* tasks. The motivation of the Next-Fit adaptation is to improve the device utilization. In this paper, EDF-NF algorithm is applied online and it is run when a new instance of a periodic task becomes ready. Complexity analysis show that it is performed in $O(n)$ time, where n is he number of tasks.

Paper [93] targets the makespan minimization of an application, described by a DAG, with partial reconfiguration. Authors compares their list-based heuristics with a Mixed Integer Linear Programming (MILP) formulation and a meta-heuristics approach based on GAs. The latter is discussed in Section 3.3.2. This paper focuses on static scheduling of applications which are described with Direct Acyclic Graphs (DAGs) by considering partial reconfiguration of the FPGA. Each task in the input graph is preceded by a reconfiguration task that models the time spent to load the bitstream onto the configuration memory. Because partial reconfiguration is addressed, authors consider tasks prefetching. Task prefetching is defined in [61] and consists in hiding the reconfiguration overhead (or part of it) by loading tasks into the physical area in advance with respect to when they are needed. This allows the FPGA to be partially reconfigured when another task is running. Architecture is modeled with a *parallel reconfiguration model*, first defined in [110] and shown in Figure 3.1.

Reconfigurable logic elements are grouped in homogeneously contiguous tiles, which are in turn composed of a reconfigurable circuit and the configuration SRAM that drives such circuit. If a task requires more than one tile, they have to be allocated in a contiguous way. All the configuration SRAMs of each tile are connected through a crossbar to a number of parallel reconfiguration controllers such that each reconfiguration controller can have access to each reconfiguration SRAM but at most one at a time. This architecture is applicable also in the case of full reconfigurable device. Tasks of DAG have a dual nature. Hardware tasks are labeled with the Hardware

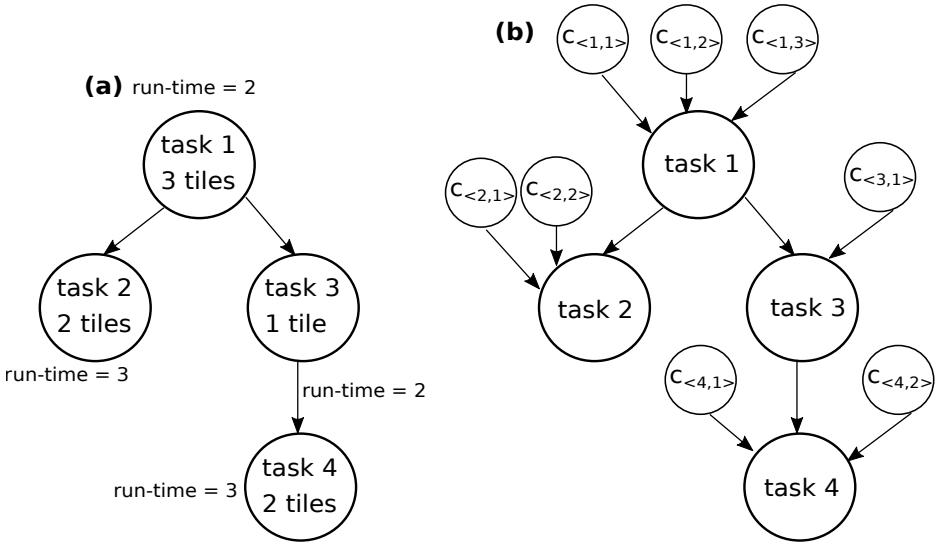


Figure 3.2: Case study from [93]

Execution time (HET) and with the required resources in terms of tiles. Configuration tasks which represent the configuration of a single tile are added to the input DAG and labeled with the reconfiguration time. Thus, if a task requires n tiles of logic elements, the corresponding node in the input DAG is preceded with n parallel reconfiguration nodes. DAG structure as well as its parameters are shown in Figure 3.2.

The challenge of minimizing the overall execution timing of an application described with a DAG is achieved through task prefetching: tasks are loaded in the reconfigurable area before they are ready, according to the availability of tiles and reconfiguration controllers. As well as any other list-based scheduling strategy, each task has a priority that represents the urgency of configuring such task onto the reconfigurable hardware. The task with highest priority is scheduled as soon as there are free resources. Priority is calculated by considering three elements, namely *mobility*, *gap* and *delay*. More precisely, mobility represents the urgency of an execution (i.e., how the current scheduling time is close to the time in which a task becomes ready to be executed), gap shows how much benefit a task can get if it is immediately selected for scheduling and delay estimates how many configurations will be delayed if the task selected for configuration cannot start immediately. The latter parameter assigns a larger delay to tasks with more successors and gives no advantages in prefetching successors while predecessors have not been scheduled yet.

The algorithm starts to iterate by setting the scheduling time $s - time$ to 1 and it concludes when all tasks have been scheduled. At each iteration, priorities for all

tasks are calculated. A free tile and a free controller are found, if they exist. The first task of the priority list is then configured if a certain number of contiguous tiles that satisfy requirements on resources are found, but it may not be ready for execution because of configuration prefetching. Thus, the ready time for execution is calculated. This process of looking for free tile-controller, scheduling of configurations and executions is repeated as long as a free pair exists. If the algorithm is not able to find a free pair tile-controller, $s - time$ in increased by 1 and a new iteration takes place. With respect to the calculus of the priority, mobility is calculated as $(ALAPs - time) - (ASAPs - time) + 1$, where *ALAP* and *ASAP* respectively mean *As Long As Possible* and *As Soon As Possible*. Gap is calculated by assuming that task is configured at $s - time$ and it can be executed at $(ASAPs - time)$, so its value is $(ASAPs - time) - (configurationEnds - time)$. Finally, delay is the normalization of the total number of successors. Each parameter is weighted with a different value not specified by authors. This list-based scheduler executes in a run-time of milliseconds for workloads similar to those used in this thesis (refer to Chapter 5) by obtaining solutions similar in quality to those produced by meta-heuristics used for comparison. Even though it targets partial reconfiguration, there are analogies between this paper and the approach used in this thesis such as the fact that a metrics takes scheduling decisions by considering the global situation (i.e., it jointly considers the current state of the scheduling, the available hardware resources and the dependencies in the input DAG). However, the calculus of priority is not compatible with our assumptions. For example, because we do not build a scheduling order by taking decisions starting from the source of a DAG, it is not relevant for us talking about *ASAP* and *ALAP* scheduling time of a task. Moreover, priority in [93] takes into account only the number of successors of a task, without weighting them with the execution time.

Closer to our FPGA scheduling problem, [64] proposes a set of list-based scheduling algorithms that target applications, described as DAGs, which are executed onto fully reconfigured devices. Two of them aim to minimize the number of reconfigurations, whereas the third one minimizes the overhead given by communications. In this work, tasks are labeled with their resources requirements only, without expliciting their hardware execution time. Edge between tasks are labelled with a communication cost expressed in terms of units of time. Because the FPGA is seen as a co-processor of a master CPU, the execution model of hardware tasks presents four different types of timing:

1. The time to entirely configure the FPGA
2. The communication time to transfer input data between the master CPU and the FPGA device
3. The hardware execution time of tasks onto the reconfigurable logic (HET)

4. The communication time to transfer output data from the FPGA to the master CPU

Configuration time and communication times are seen as pure overhead and their contribution in the final makespan shall be minimized. All the approaches presented target this minimization goal by using common principles:

1. Minimizing the number of reconfigurations stages, in order to limit the number of interactions between CPU and FPGA

2. Overlapping as most as possible the processing of tasks within a reconfiguration stage, to exploit task parallelism

The first contribution presented in [64] is named *Weight-Based Scheduling Algorithm* (WBS) and it uses a breadth-first search (BFS) to build a scheduling solution. Nodes of DAG are sorted first for their BFS level and then for their resource occupancy in increasing order. BFS level of a node is defined as the maximum distance of such node from one among the sources of the graph. Sources are in turn defined as the nodes that do not have any predecessors. After nodes are sorted, they are visited in their order and reconfiguration stages are created. When the next task of the list cannot fit the available area, a full reconfiguration of the device occurs and the next task is scheduled in another reconfiguration stage. Once all the nodes of a certain level are scheduled, the algorithm takes into account the next level. Figure 3.3a shows an example of the reconfiguration stages created by WBS algorithm. In the figure, nodes are internally labeled with their normalized resource consumption in terms of logic elements slices. This approach is very simple and fast, but it has a major default. It considers the parallelism between tasks but such parallelism is not weighted with the execution time. Avoiding to consider a timing-weighted parallelism makes inadequate the concept of BFS level: indeed, a single task may have a HET that can be strongly larger than the sum of $HETs$ of a sequence of tasks. This produces solutions that are sub-optimal whether the variance among HET of tasks is large.
The second contribution of [64] partially mitigates the fact to separately consider each BFS level. The algorithm is named *Highest Priority First - Next Fit* (HPF-NF) and it is inspired by the EDF-NF of [41], already presented in this section. HPF-NF does not consider deadlines but it focuses on reducing communication overhead between nodes. Similarly to WPS algorithm, priorities are assigned according to data-dependencies (i.e., predecessor nodes have a better priority than their successors) and according to resources requirements in an increasing order. Then, reconfigurations stages are built by following this order. With respect to WPS, HPF-NF differs when the next task in the priority list cannot fit the remaining area of the FPGA. Differently to WPS, HPF-NF does not create a new reconfiguration stage for the next task in the priority list, but it tries to schedule tasks with lower priority in the same

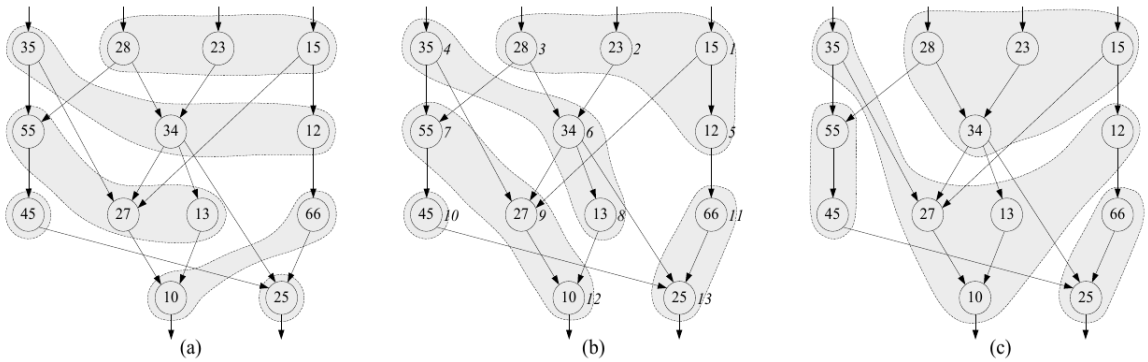


Figure 3.3: Case studies from [64]. Figure (a) shows the application of WBS algorithm. Figure (b) shows the application of HPF-NF algorithm. Figure (c) shows the application of RDMS algorithm. The latter is explained in Section 3.3.1.2.

reconfiguration stage, if data-dependencies make this scheduling feasible. This is the case of task with priority *15* in Figure 3.3b (in Figure 3.3b, the number inside the task represents the resource consumption, whereas the number on the right side of each task represents the priority calculated by HPF-NF algorithm), that is scheduled with tasks whose priority is *1*, *2*, *3* but not *4*, which belongs to the subsequent reconfiguration stage. The *Next-Fit* option allows to improve device utilization and saving reconfiguration time and this can be seen that scheduling produced by HPF-NF algorithm (Figure 3.3b) has one reconfiguration stage less than scheduling produced by WPS (Figure 3.3a). Even though each reconfiguration stage has an impact on the overall makespan, there are no guarantees that the overall makespan of the scheduling produced by HPF-NF is better than the one produced by WPS. With respect to Figure 3.3b, if the HET of tasks with priority *5* and *6* is similar and incredibly larger than HET of other tasks and to the configuration time, scheduling produced by WPS would result in a better makespan because it exploits the parallelism of those two tasks.

3.3.1.1.1 HEFT Next-Fit - HEFT is a list scheduling algorithm where tasks are scheduled in decreasing order of their upward rank, that is computed based on the critical path (in terms of hardware execution time) from a task to the sink of the dependency graph. HEFT was initially proposed for multi-processor platforms and is directly adaptable to reconfigurable platforms: tasks are assigned to *logical* processors (our slots) rather than physical processors.

- $rank_u(n_i) = w_i + \max_{n_j \in succ(n_i)}(c_{ij} + rank_u(n_j))$ is defined as *upward ranking* for a certain task n_i, from the sink (included) to n_i itself.

- $succ(n_i)$ is the set of immediate successors of task n_i (from a topology point of view).

- c_{ij} is the cost of the link between task n_i and its successor n_j. Even though the original HEFT considers this parameter, this is not directly taken into account in *Slot*.

- w_i is the computation of task n_i.

The algorithm is composed of two parts: the *task priority phase* and the *processor selection phase*.
Task priority phase: the upward ranking is calculated for each task. At a first glance, it seems a recursive approach. However, from an implementation point of view, it is enough a breadth first search. Then, tasks are ordered in decreasing order based on their calculated upward ranking. This ranking guarantees the respect of topological dependencies in the provided list. In case of equal scores, a random tie-break is done.
Processor selection phase: in the original version of the algorithm, a task n_i is assigned to the first processor p_j that is free when the task n_i starts.
Whereas the first phase could be kept using an architecture FPGA-based instead of a multi-processor platform, some minor and non-invasive modifications shall to be done in order to adapt the second phase. Basically, tasks n_i is assigned to the first valid slot RS_j. Also in this case, the politics *first-fit* is kept.

Many variants of HEFT exist in the literature. We selected the HEFT-Next-Fit as it improves the utilization of logic processors (FPGA slots). In HEFT, if a task t does not fit a logic processor p because of resource constraints, t and all higher-rank tasks are assigned to another logic processor (a new slot, in our case). In HEFT-NF, instead, p can execute tasks with higher ranks than t, as long as there are available resources.

3.3.1.2 Packing-based scheduling

Packing-based algorithms are related to Bin-Packing Problem (BPP). Bin-Packing Problem is a well-known combinatory optimization problem which is proven to be NP-hard [75]. In BPP, items of different volumes v must be packed into a finite number of bins, which are in turn characterized by a fixed volume V, in a way that minimizes the number of bins used. In the simplest version of BPP there are no constraints on items shapes. In other words, a set of k items can fit a bin if and only if $\sum_{i=0}^{k}(v_i) \leq V, 0 \leq v \leq V, k \in N*$.

Bin-Packing Problem can be, in principle, applied to FPGA scheduling. Items in BPP can be considered as tasks of an application: volume v of items corresponds to the requested resources by tasks, volume V corresponds to the overall resources offered by the FPGA and the instantiation of a new bin corresponds to a new reconfiguration stage. As described in Table 3.1, some works represent the FPGA with a 2-dimensional model, namely resources offered by the device are seen as a rectangle of reconfigurable logic elements. In this regard, resources consumption of tasks is represented in a geometrical space and the shape of tasks therefore plays an important role in the evaluation of a reconfiguration stage. Variants of BPP cover this case too. For example, 2-D BPP considers sizes of both tasks and bins in terms of finite rectangles. A set of items fit a bin not only if the sum of the items area is at most equal to the area of the bin, but also if their placement does not exceed the physical limits of the bin. Article [81] and survey [80] contain an overview of 2D-BPP problems with mathematical formulations and the most known heuristics and meta-heuristics used to solve them. Most heuristics are based on well-known heuristics such as First-Fit, Last-Fit and so on. They are not dissimilar to the list-based algorithms already discussed.

Despite the analogies, there are four important differences between the BPP and our FPGA scheduling problem, which make the solution presented for the BPP unsuitable for the FPGA scheduling problem. Firstly, in BPP items are free from any data-dependencies and the level of parallelism in BPP is consequently the maximum possible. This case corresponds to the theoretic case of a DAG with no edges, which violates our design assumptions, since we have assumed that application DAGs always present a unique source and sink. The second important difference is that items in BPP are only labeled with resource consumption, whereas in FPGA scheduling problem tasks have a hardware execution time (HET) in addition to resources. This increases the computational complexity because the introduction of a new dimension (i.e., HET) and, differently from resources, it directly contributes to the overall makespan. Thirdly, the introduction of a new reconfiguration stage has a cost on the final makespan (i.e., the reconfiguration time), whereas introducing a new bin in BPP is free of costs. Ultimately and most importantly, the main goal of bin-packing is to minimize the number of bins. In FPGA scheduling problem, this corresponds to the minimization of the number of reconfigurations which is not necessarily translated into the minimization of the makespan of an application: few high-latency stages may require more time to execute than a larger set of low-latency stages. This is possible even though a new reconfiguration stage introduces a timing overhead given by the reconfiguration.

For the sake of completeness, FPGA scheduling problem is similar to another optimization problem named Knapsack Problem [85]. A Knapsack Problem is a combinatory optimization problem. Given a set of different items, each of them being characterized by a weight and a value, the goal is determining which items to include

in the knapsack in order to maximize the value of the collected items while respecting the size constraints of the knapsack. FPGA scheduling problem looks like a Knapsack Problem, in which the FPGA area can be seen as the size of the knapsack, weight of an item corresponds to the requirements of a task in terms of hardware resources and value of an item is the HET of a task. However, solving a Knapsack Problems does not directly solve the FPGA scheduling problem addressed in this thesis. We assume that a single application needs more than one reconfiguration stage to be exactly solved and this would require to run the knapsack problem multiple times, obtaining at each step a knapsack that is hopefully the best one. Even if each knapsack problem would be solved in an exact way, there is no guarantee that this would lead to a good overall solution. This because each step aims to generate the best local knapsack without considering the overall scheduling situation as we do in *Slot*.

This is what it is iteratively done by [63] and [62], which extend paper [64]. Differently from WPS and HPF-NF list-based algorithms, presented in Section 3.3.1.1, for this contribution authors assign an HET to each node, that in the specific example of Figure 3.3c is equal to the resource consumption of the task. Authors aim to solve such FPGA scheduling problem by iteratively solving a variant of knapsack problem that considers dependent items [69]. They propose an algorithm named *Reduced Data Movement Scheduling* (RDMS) that tries to enssure that each reconfiguration stage uses the maximum combined resources while exploiting data-dependencies. Authors apply a dynamic programming algorithm, presented in [72] to solve in an optimal way this dependencies-aware Knapsack Problem, by setting the capacity of the FPGA as an upper bound constraint. This optimal dynamic programming algorithm to solve knapsack problem is run several times until all the tasks are scheduled. With respect to WBS and HPF-NF, RDMS is able to further reduce the number of reconfigurations stages, as shown in Figure 3.3c compared to Figures 3.3a and 3.3b. However, it is a greedy algorithm that takes local optimal scheduling decisions without considering the global scheduling situation. All the critical issues introduced for both the bin-packing problem and the knapsack problem are applicable here. It is true that reconfiguration constitute an issue and it should be minimized. However, the best solution can be characterized by a larger overhead because it better exploits timing-weighted parallelism between tasks.

An important family of packing-based algorithms is represented by Server-based (SB) algorithms. Server-based scheduling is a recent technique from the real-time community [32] that groups tasks in the so-called servers. In [32], a server is defined as a periodic task whose purpose is to serve requests for resources as soon as possible. Static server-based scheduling for FPGAs has been first studied in [40] for not dependent periodic tasks. Authors proposed a scheduling technique to create predictable task timing for the scheduling of periodic tasks onto fully reconfigured devices. Tasks and hardware models are the same to those already introduced

for [41]. For reminder purpose, each task is labelled with a hardware execution time HET, a period time P and the proportion $0 \leq A \leq 1$ of a single type of hardware resource (e.g. LEs) it requires. The authors propose two utilization metrics for a set Γ of tasks:

1. Time utilization factor: $U_T(\Gamma) = \sum_{T_i \in \Gamma}(HET_i/P_i)$.
2. System utilization factor: $U_S(\Gamma) = \sum_{T_i \in \Gamma}(HET_i * A_i/P_i)$.

In the extreme case where all tasks are executed sequentially $U_T(\Gamma)$ represents the fraction of time during which the FPGA is used. The $U_S(\Gamma)$ metric is a time-area product measuring the utilization of the FPGA in any scheduling, with or without parallelization. Obviously, as noted by the authors, if $U_S(\Gamma) > 1$ there is no feasible schedule for task set Γ.

Authors of [40] propose an algorithm named Merge Server Distribute Load (MSDL) that is based on the concept of a task *Server* defined as $S_i = (R_i, P_i, HET_i, A_i)$. R_i is the set of tasks for which resources are reserved, whereas the other terms have already been defined for a single task and must be applied to the set of tasks R_i. For example, A_i is the sum of area of tasks composing R_i. MSDL aims to construct a set of servers Ω starting from the original task set Γ by merging tasks together for parallel execution. Servers execution is then sequentialized. In this work, tasks are not dependent on each other but they have a deadline equal to their period. Once servers have been composed, their order will be defined by the well-known Earliest Deadline First (EDF) algorithm. Server composition works as follows. Each task is initially considered as a server composed only of the task itself. At each iteration of the algorithm, a couple of servers is chosen and merged if possible (i.e., if area of the merged server does not exceed the available FPGA area). The couple of servers is selected according to a greedy approach. For each possible couple of servers, servers which will be merged are those that minimize the time utilization of the device given by $U_T(\Omega_{old}) - U_T(\Omega_{new})$ by increasing the device utilization $U_S(\Omega_{new}) - U_S(\Omega_{old})$ and for which constraints on area are respected. The algorithm returns when no more couples candidates can be merged. Paper [42] applies a similar principle by allowing a task to be realized in alternative implementation. For example, for some type of tasks an implementation which occupies more resources needs less HET to be executed. This happens when the intrinsic parallelism of a task can be exploited by using more reconfigurable resources [71]. Authors proved that alternative implementations enhance the quality of the proposed solutions and it enhances the resource utilization of each server. This idea is confirmed in paper [106] and it can be integrated in an entension of our work. The goal of article [40] is to find a feasible scheduling while minimizing the number of reconfigurations.
This approach is not effectively applicable to applications whose tasks are connected by dependencies (e.g., DAGs). Indeed, the utilization metrics take into account

only parameters (area, timing, etc.) within a single server without considering the global scheduling situation. For instance, tasks that compose a server shall generate an high quality server (i.e., a server whose tasks well exploit the time-weighted parallelism) but also by leaving a fair amount of time-weighted parallelism among unscheduled tasks, in order to minimize the makespan contribution given by tasks which are unscheduled. The utilization metrics of [40] evaluates only the quality of a server, without considering the situation of unscheduled tasks. Thus, the best greedy choice when merging servers can lead to an inefficient overall scheduling while treating with dependencies.

Another Server-based algorithm, which schedules tasks, by considering partial reconfiguration of the FPGA, in the context of a Real-Time Operating System (RTOS), is described in [55] and [58]. Since the contribution is integrated in a RTOS, this paper deals with online scheduling of tasks that must be scheduled upon their arrival trying to respect their deadline. Features of tasks as well as their arrival order are known only at run-time. The target architecture is composed of a CPU and an FPGA but, in contrast to other works, the CPU does not only offload tasks to the FPGA but it can execute them as well. Tasks are provided both a software and a hardware implementation, and they are scheduled either on CPU or FPGA according to the decisions taken by the scheduler. The chosen scheduler is a classical EDF scheduler, but in order to minimize the rejection rate relocation of tasks from hardware to software and vice-versa is considered. With regard to our thesis, this contribution is not interesting for the scheduling strategy (indeed it is a classical EDF with no variations), but because it is applied to a RTOS context. Each task is labeled with the software occupancy (in terms of CPU workload) and hardware occupancy (in terms of required area of reconfigurable logic elements). Relocation between them is possible and relocation decisions are taken at run-time.

Work [27] targets partial reconfiguration and it relies on packing-based scheduling to minimize makespan of an application which is represented by a chain of tasks. In a chain each task has exactly one predecessor and one successor, except for the source which does not have a predecessor anche the sink which does not have a successor. This means that chains are devoid of parallelism among tasks (except for the case in which chains are executed in a pipeline way). A representative example of such chain applications is the JPEG decoding. Similarly to us, they consider an application scenario in which tasks parameters (i.e., dependencies, resources, execution timing) are known a-priori without changing or being variable at run-time. This is named *semi-online* application scenario and this assumption is taken in the contribution of this thesis too. The application can be dynamically invoked and this determines the allocated resources that are reserved to it. Authors aim to choose the right parallelism-granularity for each data-parallel task. Granularity is defined as the number of instances (i.e., copies) of the task and the HET of them. The fact to consider copies of the task is not desirable in the contribution presented in

our thesis, because it was defined to be used in a cloud data center environment: users who rent one or more FPGAs have to pay also for the resources occupied by duplicate tasks. Since they consider partial reconfiguration, they consider task prefetching too. With regard to architectural modelling, an FPGA is seen as a set of logic elements arranged in a 2D matrix, so authors take into consideration the problem of physical placements of tasks within this 2D matrix. A tuple (c_i, t_i, r_i) is assigned to each task T_i of the application graph, where c_i is the requested resources in terms of columns of logic elements, t_i is the HET whereas r_i is the time needed to charge in memory the bitstream associated to T_i. In the chain dependent tasks communicate through a shared memory, which can physically be mapped on on-chip RAM blocks or on an external memory, according to application requirements. They assume to have enough bandwidth to meet the needs of tasks at any time, as we do in *Slot* contribution presented in Chapter 4. Besides an integer linear programming (ILP) formulation, authors of [27] first introduce a heuristic to schedule simple task chains. It is named Modified First Fit (MMF) and tries to satisfy task resource constraints through a modified version of First-Fit approach. It jointly finds a physical placement and schedules tasks when trying to reduce resources fragmentation and hiding part of the reconfiguration overhead. This modified version of First-Fit relies on a local optimization that better handles the available resources. The idea is to delay the reconfiguration r_i of a task T_i, enough to save hardware resources and to parallelize the execution of T_i with the reconfiguration R_{i+1} of the following task T_{i+1} in the chain without delaying the actual execution of T_i. MFF alone does not consider data-parallelism (i.e., how the implementation of a task can be parallelizable), so authors propose PALGRAN, acronym for parallelism granularity selection, namely an application mapping approach that selects a suitable granularity of data-parallelism for individual data parallel tasks when considering key issues such as reconfiguration overhead and placement constraints. PALGRAN is an adaptation of MFF that essentially tries to greedily add multiple copies of data parallel tasks as long as it estimates that the addition of a new copy is beneficial for the overall makespan. Let's clarify this with an example. By considering a task T_2 that follows T_1 in the chain, the starting time of T_2 = end time of T_1 + r_2. However, if T_1 is 100% intrinsically parallel for all its execution (i.e., from the source until the sink), it can be divided into two tasks, whose HET is exactly $T_1/2$. Starting time of T_2 then has become $T_1/2$ + r_2. This data-parallelism concept is similar to [42], where they consider multiple implementations for the same task (e.g., one larger in terms of resources and faster in terms of HET and another one longer in terms of HET but smaller in terms of resources) and they choose the one among them that is considered the best for the overall makespan. This enhancement is combined to other considerations that have already been made for the joint placement and scheduling. However, the utilization of data-parallelism is not always a good idea because implementations which better utilize data-parallelism usually consume more hardware resources. These resources

could not be available in a certain time by consequently introducing a degradation on the starting time of a task. PALGRAN algorithm jointly evaluates all these aspects before grouping tasks. Even though paper [27] targets the scheduling by considering partial reconfiguration, which introduces an FPGA scheduling problem totally different by our FPGA scheduling problem, as explained in Section 3.1, the considerations on data-parallelism are particularly interesting and we retain we could take inspiration from them in the future work of this thesis, as explained in Section 7.2.

Paper [82] illustrates an online packing algorithm that takes into account constraints on communications among tasks as well as the physical placement of tasks. Their target is an FPGA, with partial reconfiguration, driven by a host processor. FPGA is represented with a 2D area model divided into columns of reconfigurable logic named slots. Note that, in this paragraph, the term *Slot* has a different meaning from the one used in the rest of this thesis and defined at the beginning of the chapter. Slots are a group of reconfigurable logic elements. A hardware tasks is labeled with the occupancy of resources in terms of width and height of required reconfigurable area and with four different times, namely the hardware execution time (HET), the time which is required to reconfigure the area reserved for a task, a communication time to read input data and a communication time to write output data. Tasks can exchange data through an external memory or performing read/write operations with a peripheral. Within the FPGA, the communication infrastructure is modeled in terms of communication channels, local buses, system buses, peripheral busses and their connections. Authors of [82] consider the FPGA placement problem as a 2D Strip-Packing problem [88]. Given a set of axis-aligned rectangles and a strip of bounded width and infinite height, it consists in the determination of an overlapping-free packing of the rectangles into the strip minimizing its height. When a new task arrives for scheduling, it is necessary to find a free time and space slot ($FTSS$) that can host it. In paper [83] authors designed an algorithm, named modified Flow Scanning (mFS), which returns the maximum $FTSS$ upon the arrival of a task. Paper [82] also takes into consideration communications, in order to enable the scheduler to consider a more correct data exchanges between peripherals and tasks when an application is executing. The proposed algorithm is named Communication Aware online task Scheduling Algorithm ($CASA$). In $CASA$, tasks arrive for scheduling in form of tasks chains. Exactly like [27], because they deal with chains (sequences) of tasks, data-dependencies are taken into account in the form of one single predecessor and one single successor for each task (except for source and sink which do not have respectively a predecessor and a successor) but there is no parallelism among tasks. The algorithm is logically divided into three steps: firstly, upon the arrival of a new task, mFS algorithm finds all the available $FTSSs$ by solving the strip-packing problem. Then, a control step prunes or adjusts all the $FTSSs$ that can generate a conflict with the configuration port. Thirdly, a $FTSS$ is chosen for scheduling as soon as communication requirements are satisfied. This last step

is performed through a heuristic named locked communication scheduling (LCS). For each task of the chain, a contention on a shared resource may appear. This is the case in which a task T_i wants to read/write data from/to a shared memory or a peripheral and the latter is being used at run-time by another entity. The access of task T_i to the shared resources is consequently delayed and the communication times are delayed as well. When that happens, $CASA$ algorithm adjusts the communication time for task T_i and, once the new timing has been evaluated, it will choose the most appropriate $FTSS$. The preference is a $FTSS$ which is adjacent to the $FTSS$ allocated for a task T_j having a direct data-dependency with T_i. This is the most suitable situation to minimize communication overheads because tasks T_i and T_j can exchange data by using a local bus (a bus that directly connects adjacent blocks). If that is not possible, task T_i will be scheduled in a slot that is not adjacent to the slot allocated for T_j. Then, the two tasks communicate through a *system bus*, which horizontally expand the FPGA by connecting all the local busses and allowing communication among non-adjacent slots. Finally, if this situation is not applicable as well, tasks T_i and T_j will communicate through an intermediate peripheral buffer. It is particularly interesting how authors of [82] model the communication within an FPGA and from an FPGA towards externals peripherals. Models proposed in [82] can be directly integrated in our contribution and a solver can evaluate the overhead given by tasks which exchange data over such communication infrastructure.

3.3.2 Meta-heuristics

Meta-heuristics are general optimization procedures that are independent from the specific instance of the problem. Meta-heuristics are used for large-scale problems when the solution space is too large to be entirely explored in a reasonable time. They usually lead to good solutions without ensuring that the global optimum can be found. Although they require more time than the traditional heuristics (i.e., seconds to minutes), their implementation is simple. The basic idea is to explore the solution space by avoiding local optimums, namely solutions which are optimal within a neighboring set of candidate solutions. Meta-heuristics are based on two general concepts that are common to all the algorithms of this family. The *intensification* allows a promising region of solutions to be better explored by improving the current solution. *Diversification* allows the escape from the *local optimum* trap by forcing the research towards unexplored regions. A good diversification strategy makes a meta-heuristic less dependent on the initial solution. Several contributions rely on meta-heuristics to address the FPGA scheduling problem. Because their run-time, most of them are applied to an offline scenario or to an online scenario if it is characterized by a limited size of the problem (e.g., a limited number of tasks to be scheduled). With regard to FPGA scheduling problem we mention, among the most known and consolidated meta-heuristics, Genetic Algorithms (GAs), Tabu Search (TS), Ant

Colony Optimization (ACO) and Simulated Annealing (SA).

Genetic Algorithms (GAs) [84] is a guided random search technique inspired by evolutionary biology. Starting from a population of solutions (individuals), the idea is to mute them in an iterative way by randomly combining individuals through the *crossover* operator and to modify them by introducing disorder elements through the *mutation* operator until a new population is produced. The obtained solution, named *chromosome*, is composed of a population whose quality should be closer to the optimum than the previous one. The quality of a solution is evaluated by a *fitness* function. and it is obtained by the proper *selection* of a subset of the newly generated individuals that replace a subset of the original individuals, according to the *evolution strategy*.

Paper [93] targets the makespan minimization of an application, described by a DAG, with partial reconfiguration. Authors compare their list-based heuristic with a Mixed Integer Linear Programming (MILP) formulation and a meta-heuristics approach based on GAs. We have already introduced the problem description as well as paper assumptions in Section 3.3.1.1 where we have described the list-based heuristic approach for this paper. For reminder purposes, this paper focuses on the static scheduling for dependent tasks, on dynamic reconfigurable FPGA. Tasks are labeled with the required resources R_i in terms of required tiles (a tile is a group of reconfigurable logic elements) and with the hardware execution time HET. The input DAG includes extra nodes representing the configuration that precedes the execution of a task and the architecture includes some reconfiguration controllers such that one reconfiguration controller can reconfigure one tile at a time. Then there is a description of how authors implemented the main entities such as the generation of the initial population, genetic operators (e.g., crossover, mutation, evaluation, selection) and the evolutionary strategy.

The group of initial solutions, which compose the initial population, is derived through a resource-constrained list scheduling strategy. Firstly, a *ready* task is randomly selected. A *ready* task is a task whose predecessors have already been scheduled. Secondly, a controller is randomly selected as well as the required number of contiguous tiles. Thirdly, the process is repeated until there are unscheduled tasks, otherwise an initial individual (i.e., a string that contains a sequence of tasks) is created.

Crossover operator must combine individuals when generating feasible solutions. This is achieved as follows. Two individuals, named *child1* and *child2*, are generated by combining two parent individuals named *parent1* and *parent2*. For each parent, the application graph that corresponds to the scheduling solution is divided in two DAGs named G_L and G_R (namely, left graph and right graph). G_L and G_R are built in such a way that only one edge connect G_L to G_R. Crossover sites are then selected for each parent. Strings which are associated to each parent are divided into two parts in such a way that all nodes in the left string belong to G_L and all nodes in

right string belong to G_R. Once the crossover sites have been marked in parents strings, they copy the left-string of *parent1* to *child1*. The right part of *child1* is generated from the right-part of *parent2*. An ASAP scheduler is used to schedule tasks for which there is ambiguity (i.e., tasks that are both in left-part of *parent1* and in right-part of *parent2* or neither. This allows a feasible child to be constructed.
Mutation is performed by separately considering task nodes and configuration nodes because of their different nature. With regard to tasks, a task is randomly chosen and moved to a new physical location that can guarantee the availability of the required number of tiles. Mutations concern configuration nodes for a task. If a task requires N tiles, N reconfiguration controllers will be assigned to it. Such controllers are then rotated. For example, if $N \leq 4$, controller which is assigned to tile 1 is replaced by the controller previously assigned to tile 2 that is in turn replaced by the controller assigned to tile 3 and so on until the controller originally assigned to tile 1 takes in charge the reconfiguration of tile N.
Evaluation of individuals of a population is done through a fitness function which evaluates the length of the critical path. The objective function of the genetic algorithm is the minimization of the application makespan. The size of population is kept fixed for all the iterations. 80% of new individuals survive and replace the worst individual among parents. They will escape from local optimum by dynamically mutating the mutation probability if the fitness of newly generated individuals is not far from those of the original population.
Authors tested their approach on a synthetic benchmark composed of 10 DAGs, in turn composed of 10 tasks and with few practical applications. Despite the size of the test suite, the scheduling deviation is 0.85% far from optimum, which is calculated through constraint programming. Average run-time is 0.91 s, far from the run-time achievable through heuristics (i.e., order of milliseconds for similar DAGs).

Paper [39] proposes a meta-heuristics based on Genetic Algorithms to minimize the delay of independent tasks that must be scheduled onto a fully reconfigured FPGA. The delay for a task t_i is defined as $T_{Delay}(t_i) = T_F(t_i) - T_D(t_i)$, where $T_F(t_i)$ and $T_D(t_i)$ are respectively the actual finishing time and the deadline time for t_i. In addition to these parameters, tasks are labeled with the arrival time, the hardware execution time (HET) and with a couple of values which quantify the required number of gate arrays (i.e., the resources consumption) on x and y axis of a geometrical space. This work is interesting because they improve traditional GAs with the Small World (SW) network model. A SW network model can be considered as a not oriented graph where the typical distance L between two randomly selected nodes grows in proportion to the logarithm of the number of nodes N in the graph [109]. It can be informally defined as a graph characterized by tightly direct connections among adjacent nodes and poorly remote connections. In principle, such strong *locality* character fits well with the FPGA scheduling problem targeted by authors, which is the minimization of tasks delay in a *semi-online* scenario. In the latter tasks

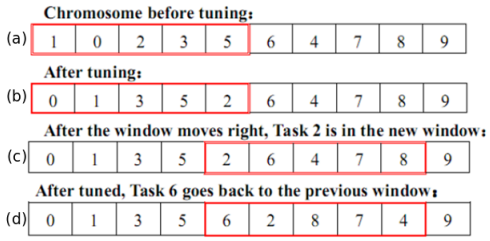


Figure 3.4: GA-SW generation of the initial population of [39]

which are characterized by a different arrival time, must be scheduled but the sequence of tasks as well as their parameter is known a-priori. Optimal solutions for such FPGA scheduling problem, which are based on authors' observations and experiments, also have a strong locality character, in the sense that the best sequence (i.e., the sequence of tasks that minimize the overall delay of the set) is close to the deadline-based sequence. Because of this locality analogies, the idea beyond this paper is to use the SW network model to generate the initial population for a Genetic Algorithm. Generally speaking, the quality of the initial population is an important factor for a GA: if a population which is very far from the region of the optimal solution is provided, several iterations will be necessary to eventually move the newly generated individuals to the right region. Moreover, the solution domain exponentially increases if the number of tasks increases as well. The motivation of a GA-SW algorithm is primarily to reduce the initial population domain by maintaining a good quality. Secondly, authors re-designed crossover and mutation operators to make them adapted and effective with the SW model.
The initial population of GA is generated as follows. Firstly, a sequence of tasks is sorted according to the deadline of tasks. Such sequence is later *fluctuated* by utilizing a gliding window in order to enable the newly generated individuals to keep the advantages given by the locality principle and not to deviate too much from their original position in the deadline-sorted sequence. The gliding window has size m, with m less than the total number of tasks n, it starts from the left of the sorted sequence and it moves right until all tasks in the original sequence are considered. Figure 3.4 illustrates an example of this process. The sorted sequence is composed of 10 tasks ordered through the earliest deadline first paradigm, whereas the size of the gliding window is fixed to 5 tasks (Figure 3.4a) . Tasks within the window are tuned according to a certain probability (Figure 3.4b), then the window is moved right for 4 positions (Figure 3.4c) and the involved 5 tasks are tuned in a similar way (Figure 3.4d).

Conventional crossover and mutation operators are adapted to the SW model. Their adaptation has the purpose to reduce the number of unfeasible solutions and speed-up the convergence of the algorithm. Instead of the traditional crossover, which generates a child individual by mixing parts of two parents, with the adapted method, named Converging Crossover (CX), a child exactly imitates only a part of a parent. With respect to mutation, traditional single-point mutation can generate unfeasible solutions, so authors designed a small-world two-points window-based exchange mutation operator (STWEMO) which generates a higher number of valid individuals. Firstly, the probability of a task to be moved to the location of another task (and vice-versa) is inversely weighted with the distance of the two sequence in the original deadline-sorted sequence. An exchange of far tasks can dramatically decrease the quality of the solution though. So they are confined withing a window in order to better utilize the advantages offered by the locality principle, especially when the number of tasks is large (always according to the described probability principle: the more two tasks are close in the original deadline-sorted sequence, the more likely their exchange will occur). Authors proved that the complexity of GA-SW is $O(n^2)$, where n is the number of tasks.

Paper [67] compares Genetic Algorithms and Simulated Annealing for a mixed FPGA placement-scheduling problem. GA and SA are used to calculate a physical placement of tasks onto FPGA, whereas the resulting makespan and the power consumption are compared. An FPGA is seen as a 2-D array of *clusters*, that are groups of logical elements. Each task (named design clusters occupies a sub-matrix of clusters. Given a starting schedule, SA iteratively and randomly swaps design clusters. Some of these swaps improve the resulting scheduling, some others introduce a degradation and from a certain iteration they are not be accepted. Keeping non-improving solutions allows to escape from local optimums. Authors tested a GA whose fitness function takes into account execution timing of tasks as well as the delay of connections between tasks, wiring cost (i.e., the amount of resources required) and power estimation. Chosen GA algorithm is based on a fine-grain genetic mutation approach in which crossover operation is not used and mutation randomly swaps the position of design-clusters. This choice has been done to keep as many analogies as possible with SA and consequently compare the efficiency of *similar* algorithms. Indeed, both SA and fine-grain genetic mutation based GA are based on the same operation, the random swap of clusters: using a GA that includes the crossover operation would have disrupted the experiment. Authors concluded that fine-grain genetic mutation based GAs are not as good as SAs.

Paper [49] proposes a joint mapping-scheduling of tasks and communications onto a target architecture by addressing makespan minimization. To do that, they propose am approach based on Ant Colony Optimization (ACO). Ant Colony Opti-

mization is a meta-heuristic based on a stochastic decision process that was initially introduced to solve another well known optimization problem that is the Traveling Salesman Problem [44]. ACO has this name precisely because it takes inspiration from the collaborative behavior of ants while searching for food: first, each ant leaves the nest taking a random direction. Throughout a path towards a source of food, each ant releases a trace of *pheromone*, a secretion that triggers a social response in other members. Other ants are then motivated to follow a path with pheromone with a probability that depends on the amount of pheromone present in such path. Quantity of pheromone released by an ant evaporates as time goes by. Consequently, when a source food is found, the shortest path from the nest to it would be characterized by a larger quantity of pheromone rather then longer routes: other ants are motivated to follow the shortest path. ACO meta-heuristic is based on this principle: different decisions represent the different routes towards the food. Pheromones are represented through a matrix that stores, for each decision, the probability that such decision lead to a good overall solution. At the beginning of the algorithm probability is uniformly distributed among decisions. Then, iteratively, ants (that represent functions of the algorithm) construct different solution by reinforcing the most promising decision by modifying values in the matrix of pheromones. At the end of each iteration, older pheromones contributions evaporates by reducing some values in the matrix itself. ACO techniques are strongly used for problems in which a good solution can be obtained by taking, at each iteration, subsequent decisions. Authors in [86] demonstrated the superiority of ACO with respect to Tabu Search, Genetic Algorithms and Simulated Annealing for the resource-constrained scheduling problem (RCSP) mentioned in Section 3.1.1.

Actually, paper [49] targets general Multi-processor Systems on chip (MpSoc) composed of a set of process elements, each of them equipped with its own local memory, connected through a set of communication elements (e.g., a system bus in the simplest representation). We can apply this architecture to the FPGA scheduling problem we target in this thesis because authors classify hardware resources in two different classes, namely *renewable* resources (i.e., resources that return available after they are used) and *non-renewable* resources (i.e., resources whose quantity reserved to a tasks cannot be immediately reused). The latter is exactly the case of an FPGA: when FPGA area is assigned to one or more tasks, it returns available only upon a reconfiguration that introduces a timing overhead. We discuss this work by restricting it in a single FPGA context. In this work, authors consider resource sharing, namely the case in which a hardware resource is used by different tasks. Communications introduce an overhead on the total makespan and they are taken into account as well. In this respect, tasks are assumed to read input data and to write output data from their own local memories. Communications between tasks mapped on the same processing element are considered negligible, whereas between processing elements a time consuming, directly proportional to the quantity of

data, Direct Memory Access (DMA) is modeled. Applications are modeled through DAGs and also in this case different implementations per tasks are considered. However, resource consumption and execution timing are assumed to be statically predicted and they does not change run-time, whereas communications contribution on the makespan can be evaluated only once the definition of a scheduling order. The term *implementation point* is used to define a particular combination of resources and time required to execute a task on a certain hardware component, for example an FPGA reconfiguration stage.
Authors considers separately the construction of a solution and its evaluation. The latter can indeed be done only by knowing when incoming data transfers have been terminated. The proposed algorithm works as follows. First, an initial and feasible solution is initialized, for instance by assigning each task to the same processing element with reconfigurations injected when a sequence of tasks finishes the available resources. The makespan of this initial solution is used to initialize the best solution and the pheromone values as well. Each ant is also initialized with tasks without precedences (i.e., source tasks). After such initialization, the first colony of L ants is run. At each iteration of the algorithm, for each ant, a task is selected and it is assigned to an implementation point. Assigning a task to a hardware resource contributes to unlock direct dependent tasks that become eligible for scheduling. Incoming communications for unlocked tasks can now be evaluated and unlocked tasks become eligible for the scheduling at next iteration. This process is repeated until the whole set of tasks and communications are assigned to an implementation point. Then, the solution is evaluated from a makespan point of view: if this solution is better than the current best, it replaces the best one. If the exploration is not finished (i.e., the maximum number of generations is not reached), pheromones are updated and a local optimization is applied to the current best solution according to pheromones values. In particular, tasks part of the solution change the position, in a feasible way, according to a probability that depends on pheromones values. Pheromones are represented through a matrix that stores, for each combination of task and a feasible implementation point, the probability that such correspondence leads to an overall good solution. Matrix is updated with a metric based on the results of the choices taken by each ant with respect to the resulting makespan. This metric has been designed in order to privilege combinations for which the task is completed as soon as possible. At this step a new colony of ants is launched and the best found solution is returned at the end of these iterations. ACO is compared to a ILP formulation as well as SA, TS, GA implementations. ACO is demonstrated to be superior than each of them while keeping the run-time comparable (order of seconds) for test-cases similar in dimensions to those of benchmark used in Chapter 5.

Ref.	Category	Rec. Type	Goal	Model	Dep.	Year
[41]	LB	F	respect deadlines	1D	No	2005
[40]	SB	F	min(nb rec.)	1D	No	2007
[96]	LB	P	min(makespan)	1D	Yes	2020
[43]	MILP	P	min(makespan)	nD	Yes	2015
[93][a]	LB	P	min(makespan)	1D	Yes	2007
[93][b]	MILP	P	min(makespan)	1D	Yes	2007
[93][c]	MH	P	min(makespan)	1D	Yes	2007
[64]	LB	F	min(nb. rec.)	1D	Yes	2008
[62]	PB	F	min(nb. rec.)	1D	Yes	2009
[63]	PB	F	min(nb. rec.)	1D	Yes	2010
[42]	PB	F	respect deadlines	1D	No	2006
[55]	SB	P	feasible sched.	1D	No	2006
[58]	SB	P	feasible sched.	1D	No	2009
[39]	MH	F	min(tot. delay)	2D	No	2010
[67]	MH	NaN	min(makesp-power)	2D	No	2010
[79]	MILP	NaN	min(mkspan + IO)	2D	Yes	2009
[82]	PB	P	min(makespan)	2D+1	Yes	2010
[51]	MILP	F	min(nb rec.)	1D	No	2008
[27]	PB	P	min(makespan)	2D	Yes	2009
[112]	MILP	NaN	min(makespan)	1D	No	2018
[102]	LB	P	max(acceptance)	2D	No	2004
[49]	ACO	F	Multiple	nD	Yes	2010
***Slot* [28]**	PB	F	min(makespan)	nD	Yes	2020

Table 3.1: Resume of related contributions

3.4 Conclusion

In this chapter we have described the related work on scheduling of applications onto FPGAs. FPGAs are complex and for this reason we have firstly explored how scheduling-relevant parameters of FPGAs are abstracted by models which represent the input for scheduling strategies. With regard to this thesis, we target the makespan minimization of applications which can be represented through a Direct Acyclic Graph onto fully reconfigured FPGAs. This situation is common in a cloud data center environment [112]. However, an FPGA scheduling problem can be different depending on objective of scheduling, input applications and FPGA character-

istics. For instance, the minimization of the number of reconfigurations, the respect of deadlines or the finding of a feasible scheduling can all be objectives of the related work. In addition to the objectives, there are FPGA scheduling problems which considers partial reconfiguration. Finally, the context where existing works are applied contribute to differentiate the related work. In this thesis, we focus on cloud data centers, but FPGA scheduling problems are widely present also in the context of Operating Systems/Real-time Operating Systems (RTOS), such as [102], [55] and [58].

However, the related work described in this thesis is firstly divided according to the used approach. We have identified three main methodologies, namely:

- Exact formulations, such as Constraint Programming, Mixed-Integer Linear Programming and so on.
- Meta-Heuristics (MHs), such as Genetic Algorithms, Tabu Search, Simulating Annealing and Ant Colony Optimizations.
- Heuristics
  - List-based heuristics, namely heuristics which assign a priority to each task of the application, then they build an ordered list of tasks according to their priority and then they take scheduling decisions according to the order of this list.
  - Packing-based heuristics, namely heuristics which take scheduling decision on groups of tasks.

In this chapter we have not dedicated a whole section to exact formulations. Works based on exact formulations, such as [51], [112], [79], [43] and [93] (the latter compares MILP formulation with a list-based heuristic and a meta-heuristic) return an optimal solution at cost of a very large run-time. We think that, with respect to this thesis, describing a set of mathematical formulations is not interesting at all and we have therefore focused on different and more original heuristics and meta-heuristics strategies. However, Section 5.2 proposes an exact formulation of our FPGA scheduling problem, which will be used to calculate the optimum of each test-case of the synthetic benchmark that we have used to evaluate the quality of our contribution in Chapter 5.

Our contribution is a heuristic, specifically it is part of packing-based heuristics. Although the related work focuses on several different FPGA scheduling problems, works described in Section 3.3.1 are comparable from several points of view, such as run-time, complexity, strategy and so on. This is not directly applicable to meta-heuristics, which generally return solutions characterized by a good quality, but with longer run-times than those of heuristics. The cited meta-heuristics include Tabu Search, Simulated Annealing, Ant Colony Optimization but each work includes at

least one implementation of Genetic Algorithms. In the whole section, we compare our heuristic with such meta-heuristics by only comparing the run-time. We take as a reference Genetic Algorithms. Complexity of Genetic Algorithms depends on the genetic operators, on their implementation (which may have a very significant effect on the overall complexity), on the representation of individuals and population and on the fitness function. Given the most common choices (point mutation, one point crossover, roulette wheel selection), a Genetic Algorithms complexity is $O(g * (2 * nm + n))$ with g the number of generations, n the population size and m the size of individuals. Their execution (and execution of MHs in general) is often limited to a certain threshold in timing or to a certain number of iterations in which the algorithm does not produce improving generations. With regard to MHs, the comparison of only the theoretical complexity can be misleading and for this reason, in Section 3.3.2, we have retained that the measurement of the run-time for workloads which are similar in size to those considered in the evaluation of our contribution (ref. Chapter 5) is a better way to compare the related works with our contribution.

However, we have identified some common drawbacks of existing works applied to our FPGA scheduling problem. On the one hand, existing works privileges either run-time (e.g., heuristics) or quality (exact formulations, meta-heuristics), whereas our contribution merges both sides. On the other hand, FPGA models of existing works capture only a subset of parameters (e.g., number of logic elements). We think that it has a strong impact on the overall makespan of a scheduled application and for this reason we enhance existing models by representing an FPGA with a nD-size array of resources.

Chapter 4

Slot

4.1 Introduction

This chapter presents the *Slot* heuristic, our proposal to solve the FPGA scheduling problem. We will first explain our approach to solve the problem and then justify why it is promising. In Chapter 5 we will experimentally evaluate *Slot* and compare it with exact solutions and another well-known heuristic.

Section 4.2 lists the assumptions we make about applications, tasks and target FPGAs. Section 4.3 motivates the need of a heuristic and explains why exact solving techniques are not suitable. Section 4.4 focuses on the abstract modelling of FPGA hardware resources and of the requirements of applications and tasks. Section 4.5 discusses our target platforms, how applications can share the FPGA resources of these platforms and how *Slot* can be used to minimize the applications' makespans. Section 4.6 is dedicated to the description of the *Slot* heuristic we propose. Section 4.6.5 discusses the complexity of the algorithm. Section 4.7 concludes the chapter.

4.2 Assumptions

These include restrictions that limit the domain of the problem. Indeed, FPGA scheduling has a large spacial complexity, where different nature of inputs, applications, and reconfiguration types result in very different problems that require different techniques to be solved.

The following assumptions are mainly dictated by the context of cloud data centers. Here, FPGAs are available for multiple users as a general-purpose reconfigurable platform for different types of workloads. Scheduling is thus possible under some some reasonable and acceptable restrictions on the input workloads. We have

categorized the assumptions in three families: assumptions on **fpga**, on **applications** and on **tasks**.

Assumptions on applications

- For each workload, a user disposes of one or more bitstreams that are designed off-line either by the user or are available as part of a library developed by third-parties, e.g., the cloud provider, the FPGA manufacturer.

- As users rent cloud resources, they are always aware of their workload characteristics on a target family of FPGAs. In other words, run-time or resource occupancy of tasks is known beforehand, typically thanks to data available during the synthesis and simulation of a workload bitstream, profiling or interpolation and curve-fitting from historic data. This is coherent with real cloud instances and with the the IaaS, PaaS and SaaS paradigms.

- We suppose that each application cannot execute in its entirety on a given FPGA (and the latter must be reconfigured at least once). Indeed, *Slot* algorithm shows its benefits for applications which present parallel tasks and which cannot fit the FPGA at once.

- Without loss of generality we assume that application DAGs have a single dummy source and a single dummy sink, not mapped onto the FPGA. This is to ease the design of solving algorithms by guaranteeing that each application has a unique entry and exit point. If an application has more than one source or sink, we connect all the real sources or sinks to a single dummy source or sink. These dummy sources and sinks consume no FPGA resources their run-time is zero.

- We consider acyclic applications which can be represented by Directed Acyclic Graphs (DAG). Iterative applications where outputs of one iteration are used as inputs of the next one can still be scheduled by *Slot* by computing the schedule on a single iteration and chaining identical schedules, at the cost of the potential inter-iterations parallelism that this approach cannot exploit.

- We do not allow tasks preemption. This a quite common assumption with hardware tasks where preemption involves complex and costly mechanisms and are frequently considered as inefficient.

Assumptions on tasks

- Tasks require a fixed amount of n resources and have a fixed execution time. Thus, we do not consider moldable nor malleable tasks, differently from [49] and [40]. Characteristics of tasks are known at design time.

- As a direct consequence of the previous point, we assume that the demand of resources requested by tasks **cannot vary run-time**. Few works address this case by proposing a Machine Learning model to estimate the real demand for resources of an application whose consumption varies at run-time [31].
- The time to read, write and transfer the input/output data for a task t in different memory locations is included in the hardware execution time of the task, and all communications perform without memory contention.
- All the tasks are released at the same time instant, each of them with a deadline that is equal to the deadline of the entire workload (i.e., the time granted to a user to dispose of the FPGA).
- We require tasks to be implemented without pipelining between a producer and a consumer tasks. This means that a task must completely execute before its direct successor(s) can start. In workloads that do not respect this constraint, pipelined tasks must be merged to a single task in the input dependency graph.

Assumptions on FPGA

- FPGAs are captured in an abstract way considering both their resources and processing capabilities. This is further discussed in Section 4.4.
- Characteristics of FPGAs that are going to be rented are known by users.
- We have categorized the nature of resources offered by the FPGA (or by the platform that includes the FPGA) in two families: scheduling-independent resources and scheduling-dependent resources. Details of those families are described in Section 4.4.
- The time to transfer a reconfiguration bitstream is included in the FPGA total reconfiguration time T_R.

4.3 Unfeasibility of an exhaustive analysis

This has already been partially covered in Chapter 3, in which we listed run-times of exact solvers and meta-heuristics. We also study a practical example. Exact methods (such as brute force) are unfeasible for scheduling a realistic number of tasks at run-time. Yet, in Chapter 5, we can afford to compare our heuristic algorithm with the optimum because the target test benchmark also contains relatively small applications, composed of 6-to-15 tasks applications. For 16-to-30 applications, this will not be possible. We define a topological order of a DAG as a linear ordering of its nodes such that for every directed edge uv, from node u to node v, node u comes

before node v in the ordering. Data dependency between tasks strongly reduces the number of topological orders. Consequently, the space of the solutions is reduced too, making an exhaustive analysis possible if the number of nodes is limited.

Let us consider a graph composed of dummy source and sink and n parallel tasks between them, as shown in Figure 4.1(a). The maximum number of topological orders for this graph is $n!$, not included. $n!$ is not included because, if all tasks can fit at the same time in the FPGA, the scheduling solution is trivial. Exhaustive research has to build all the possible valid solutions starting from each topological order. To do that, it has to interleave tasks with reconfigurations without ignoring any case. In fact, the best solution is not always the solution that contains the minimum number of reconfigurations. Similarly, the slots of a best solution may not be best-fitted from area point of view, as we already demonstrated in the example in Section 1.4.

As a further demonstration, let us consider the simple case in Figure 4.1(b), composed of four tasks of which two of them are parallel. In this example we assume that each task occupies exactly half of the FPGA resources. Two possible scheduling solutions are:

1. $[R, \{A\}]$, $[R, \{B, C\}]$, $[R, \{D\}]$
2. $[R, \{A, B\}]$, $[R, \{C, D\}]$

If the relation $min(HET_B, HET_C) > T_{Rtiming}$ is verified, the solution characterized by the minimum makespan is the first one, even if it has more reconfigurations. This is to say that an exhaustive analysis shall evaluate all valid combinations that interleave reconfigurations between set of tasks, for each topological order, and for all possible sets. In the simple example of Figure 4.1(b), we have two topological orders:

1. A, B, C, D
2. A, C, B, D

And, starting from each topology order, there is a total of 9 possible and valid solutions:

- $[R, \{A\}]$, $[R, \{B\}]$, $[R, \{C\}]$, $[R, \{D\}]$
- $[R, \{A\}]$, $[R, \{B\}]$, $[R, \{C, D\}]$
- $[R, \{A\}]$, $[R, \{B, C\}]$, $[R, \{D\}]$ or $[R, \{A\}]$, $[R, \{C, B\}]$, $[R, \{D\}]$
- $[R, \{A, B\}]$, $[R, \{C, D\}]$
- $[R, \{A, B\}]$, $[R, \{C\}]$, $[R, \{D\}]$

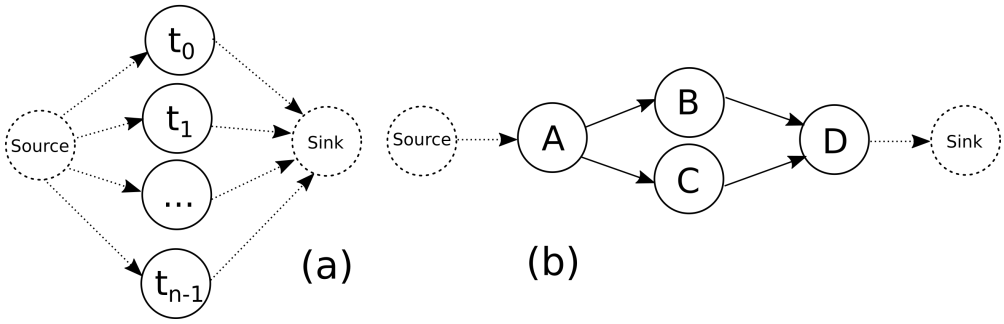


Figure 4.1: Application DAGs - simple case studies

- $[R, \{A, C\}]$, $[R, \{B\}]$, $[R, \{D\}]$
- $[R, \{A\}]$, $[R, \{C\}]$, $[R, \{B\}]$, $[R, \{D\}]$
- $[R, \{A\}]$, $[R, \{C\}]$, $[R, \{B, D\}]$
- $[R, \{A, C\}]$, $[R, \{B, D\}]$

Thus, there are 10 possible solutions to be evaluated for an example of 4 tasks only, with a maximum parallelism of 2. It is natural to imagine that this number tends to explode with the increasing of number of tasks, especially with applications characterized by a huge parallelism due to the low number of data dependencies as the extreme general case in Figure 4.1(a). Starting from the $n!$ topological orders of this graph, for each of them we need to interleave tasks with reconfigurations whenever possible, to be sure to consider all valid cases. Yet, calculating an upper bound of the complexity is not a trivial problem, because we cannot establish **in a general way** how data dependencies and resources consumption will impact the number of slots or the number of tasks for each slot. In an experimental way, however, we understood that the complexity tends to be exponential with the number of tasks. By profiling the code for the benchmark evaluated in Chapter 5, which includes 6-to-30 task graphs, for some test cases, we evaluated up to tens of thousands theoretical candidate solutions. Thus, a heuristic is absolutely necessary to scale with the problem size.

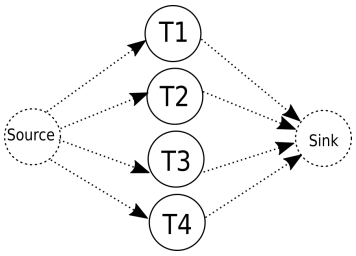


Figure 4.2: 6-tasks application DAG

4.4 Abstracting hardware resources and applications

In this regard, we have classified hardware resources according to their nature. Indeed, requests for some types of resources may have an impact on requests of other tasks scheduled in the same slot.

Different nature of hardware resources: Hardware resources have a different nature and they can therefore impact the scheduling in a different way. A little part of our contribution consists in the differentiation between *scheduling-independent* and *scheduling-dependent* resources, which are two categories that, to the best of our knowledge, have not been identified, so far, by existing taxonomies on resource constrained scheduling problems [60].

To explain how they differently impact the scheduling, let us take as a reference a DAG composed of 4 parallel tasks plus artificial source and sinks, such as the one in Figure4.2.

A **scheduling-independent resource** is a resource which is exclusively assigned to a task for the entire lifetime of the slot that contains the task. Thus, requests for a given scheduling-independent resource (such as LEs) within a slot can be handled simply by adding them together. Yet, if the total amount of requested resources exceeds the availability (for a given FPGA) or all the tasks requesting the resources cannot be part of the slot. This is shown in Figure 4.3, in which task $T4$ cannot be part of the slot that already contains $T1$, $T2$ and $T3$.

A **scheduling-dependent resource** is a resource which is assigned to a task for the entire task's lifetime. The latter is always less or equal the entire lifetime of the slot that contains the task. Tasks whose requests of a scheduling-dependent resource exceed the physical limit for a given FPGA, such as $T1$, $T2$ and $T3$ in Figure 4.4,

Task	LEs [%]	I/O bandwidth [%]
t_1	15	33
t_2	25	37
t_3	40	40
t_4	50	40

Table 4.1: Parameters of tasks for the example of Figure 4.2 - Occupacy represents the percentage of the hardware resources used by tasks in a given FPGA

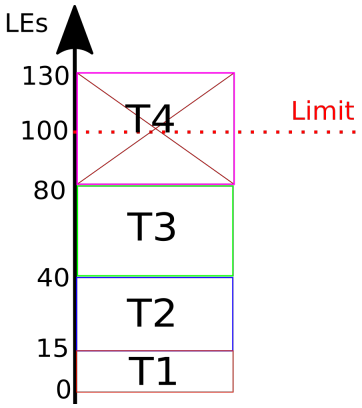


Figure 4.3: Occupancy of the FPGA with regards to the number of used Logic Elements (LEs). Tasks $T4$ cannot be part of the same slot of $T1$, $T2$ and $T3$.

cannot be executed **in parallel**. However, differently from scheduling-independent resources, they can be part of the **same slot** by introducing a delay, as shown in Figure 4.5.

Our formulation presented in Section 4.6 efficiently considers scheduling-independent resources. Scheduling-dependent resources can be treated as if they were scheduling-independent at the price of a pessimistic output schedule. Note that, to efficiently treat scheduling-dependent resources, the data dependencies in the task graph of a slot are no more a sufficient condition to determine a total execution order for tasks in such slot. Differently from hard-constraints imposed by physical limitations on scheduling-independent resources, constraints on scheduling-dependent resources can be relaxed by modifying the execution order of tasks which are part of the same slot in a stronger way than constraints imposed by data-dependencies, at cost of introducing a delay.

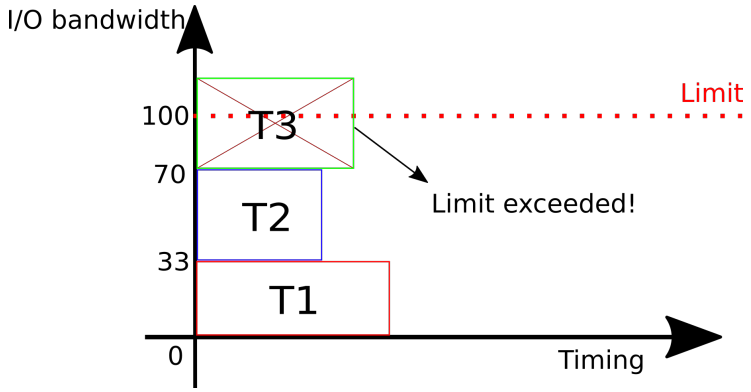


Figure 4.4: Occupancy of the FPGA with regards to the required I/O bandwidth. Tasks $T3$ cannot run in parallel with tasks $T1$ and $T2$

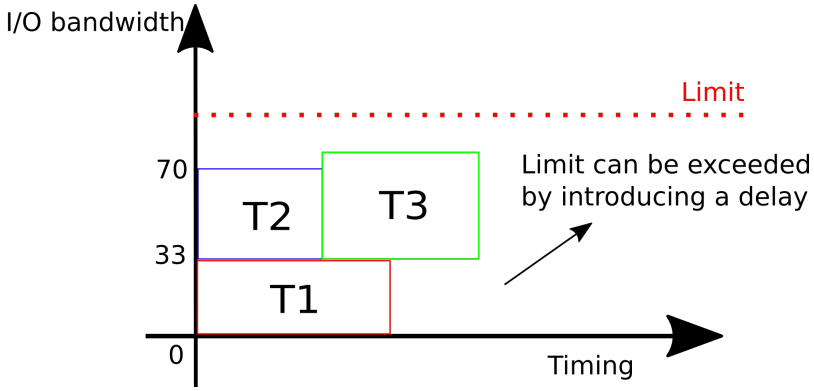


Figure 4.5: Occupancy of the FPGA with regards to the required I/O bandwidth. Tasks $T3$ cannot run in parallel with tasks $T1$ and $T2$, but they can be part of the same slot by introducing a delay

Abstracting FPGAs: FPGAs, such as those deployed in cloud data centers, offer multiple types of physical resources to allocate tasks. The basic units of an FPGA are blocks of reconfigurable logic, which we refer as logic elements (LEs). However, as we saw in details in Chapter 1, many other resources are available. Indeed, FPGAs also offer pre-built digital signal processing blocks (DSPs), e.g., multipliers, to save on the usage of logic units and accelerate workloads such as scientific computing and signal processing. Memory resources are available, to store temporary results or communicate between tasks, in the form of Random Access Memory (RAM), both on-chip (e.g., Embedded Memory Blocks - EMBs) and off-chip (e.g., Dynamic-RAM). We retain that modelling which only represent an FPGA as a bunch of LEs, like several related works do as shown in Table 3.1, are extremely limiting. For instance, let us consider an application whose tasks perform several I/O operations.A modelling

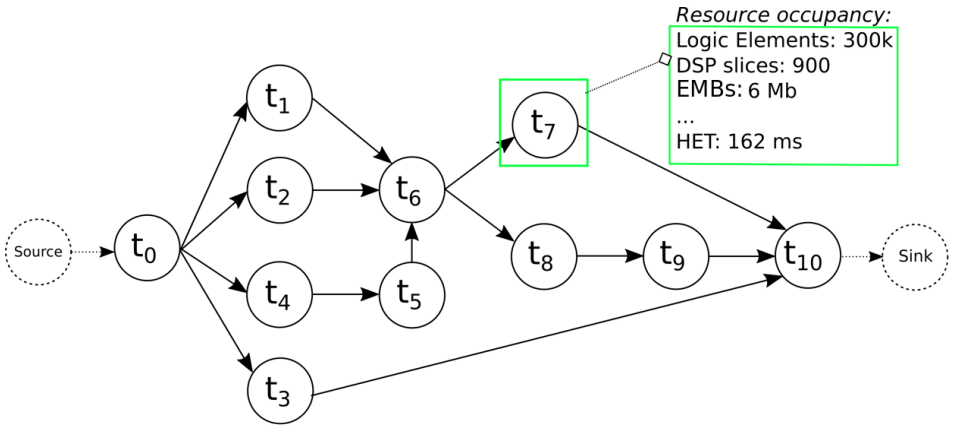


Figure 4.6: An example of application DAG which is the input of *Slot*: each node, representing a task, is labelled with resource requests information and HET

which represents the FPGA only in terms of LEs only may suggest that a certain set of tasks may share the area of the FPGA, whereas the constraints on the number of I/O pins make actually non-valid the scheduling of such set of tasks (in other words, such tasks require more I/O pins than those actually available onto the FPGA).
In addition, resources are available in different quantities, packaged in different ways (e.g., granularity of blocks, for instance rows/columns of LEs) and with slightly different denominations according to the FPGA manufacturer and/or FPGA family.
We designed *Slot* heuristic to be generic and valid for k-dimensional models of resources whose requests are constant in time and do not depend on the scheduling of tasks. Users of the algorithm (i.e., both cloud users and cloud providers) are free to model an FPGA with any type of resource, each of them packaged in any way.
Modelling of application shall be coherent with modelling of the FPGA. As shown in Figure 4.6, each task of the application must be provided with the information about Hardware Execution Time (HET) and requests of resources. The latter must be exactly the same which composes the k-dimensional model provided for the FPGA.

4.5 Sharing of applications over platforms

In our work, we target **platforms** as such in Figure 4.7, which are composed of two *logical* parts: a static region and a reconfigurable region, interconnected by a bus-based infrastructure. The static region executes on a general-purpose processor, in charge of running the reconfiguration management, source and sink tasks and a reconfigurator device that internally reconfigures the system at run-time. The reconfigurable region is composed of a reconfigurable hardware device that is assigned to one or more users by a network orchestrator (e.g., according to some service-

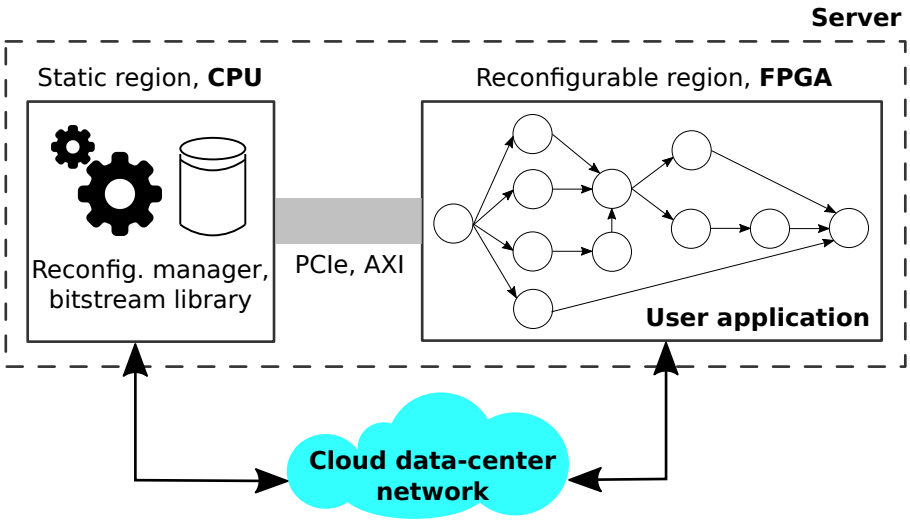


Figure 4.7: The architecture of a modern FPGA-based server

level policy). This assignment is fixed for the entire execution of any workloads. We distinguish between two different FPGA usages:

1. *Single application:* a user wishes to execute a single application onto a FPGA. In this case, there is no sharing and the whole FPGA is reserved to a single application. User can rely on *Slot* heuristic to find a scheduling that shall minimize the latency time (i.e., makespan) of the target application. This is exactly what is reproduced in Figure 4.7.

2. *Multi application:* this is the case in which several applications, possibly belonging to different users, have to execute on the same FPGA area. We further divide this case in two different sub-cases, according to policies of cloud data center providers.

    a) According to [112], there are families of cloud applications which aggregate batch of tasks before accelerating such batch. This situation is represented in Figure 4.8, in which the red application is batched with the blue application by connecting each entry node to a pseudo source node (i.e., the left black-dotted task) and each termination to a pseudo sink node (i.e., the right black-dotted task). Thus, the objective function is the makespan minimization of the entire group of applications and this can be achieved through *Slot* heuristic. With respect to *Slot*, each task of the red application is topologically parallel to each task of the blue application.

    b) An alternative policy is shown in Figure 4.9, in which an FPGA is statically divided in several reconfigurable regions, and one or more reconfigurable

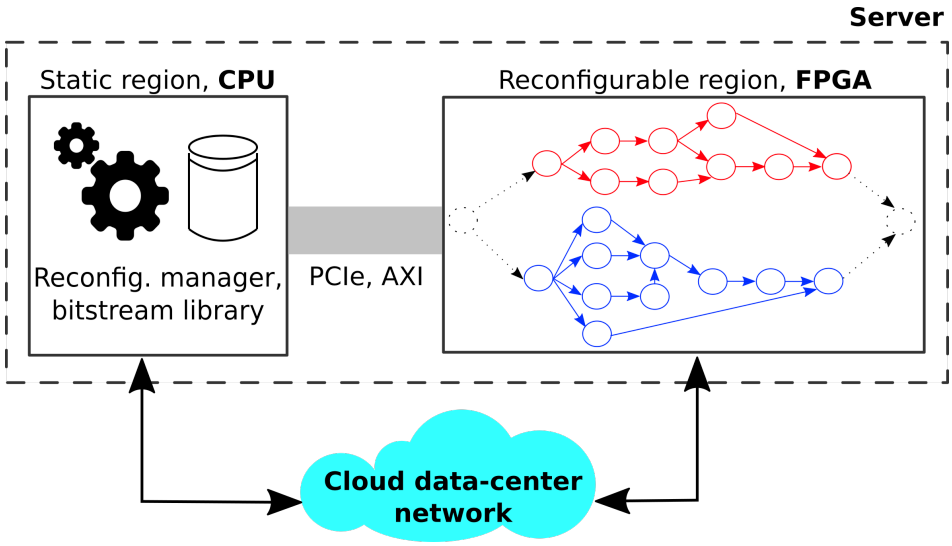


Figure 4.8: Sharing of applications on the same platform

regions are assigned to a different user/application. In the example, up to 9 applications belonging to 3 users can share the FPGA and *Slot* algorithm would be applied to each reconfigurable region. This case is not very dissimilar by the *Single application* case, but here each application sees only a part of the FPGA. However, directly considering this case would require the introduction of the partial reconfiguration to independently reconfigure each region. We deepen discuss this case in the future work, in Chapter 7.

4.6 *Slot*

First of all, we will introduce an example which will serve as a case study to better understand the algorithm, step-by-step. Section 4.6.1 shows and describes the pseudo-code of the algorithm. Sections 4.6.2, 4.6.3 and 4.6.4 deepen the three main components of the algorithm, namely the *candidate generation*, the *score* and the *final optimization phase*. Details on such three components are separately treated to keep the explanation of the main loop of the algorithms as light as possible.

The **formulation** of our heuristic is generic and valid for k-dimensional models of resources whose requests are constant in time and do not depend on the scheduling of tasks. We consider a set K that contains k resources, available in R_k units.
A **user's workload** is denoted as a DAG $G =< T, E >$ and it is executed onto platform illustrated in Figure 4.7. The source and sink tasks (control tasks), t_0, t_{n+1}, are mapped to the static region of the platform. Remaining data-intensive tasks

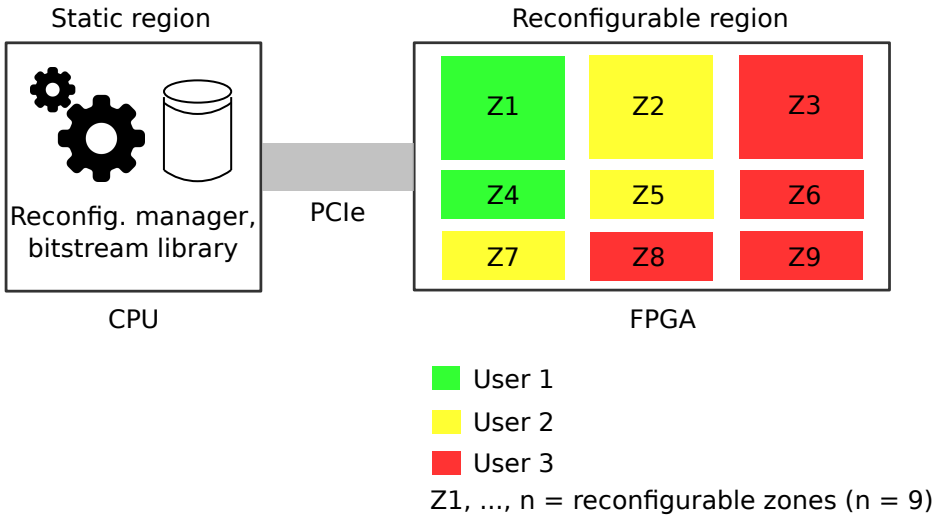


Figure 4.9: Sharing of the platform among different users through reconfigurable regiorns

$t_i \in T \setminus \{t_0, t_{n+1}\}$ are mapped onto the reconfigurable region, which is a technologically *mapped* netlist implementing the i^{th} task. We characterize it by means of a tuple $(h_i, r_{i1}, r_{i2}, ..., r_{ik})$, where h_i is the hardware execution time (HET) that is the time taken by t_i to execute. The reconfigurable resources that t_i requires are expressed by the generic tuple $(r_{i1}, r_{i2}, ..., r_{ik})$. In this chapter we adopted a 3D model in which r_{i1} is the number of logic elements, r_{i2} is the amount of (on-chip) Embedded Memory Blocks (EMBs), r_{i3} is the number of DSP blocks. Note that, for instance, for easier partial bitstreams composition, the logic elements resource could very easily be replaced by entire rows of logic elements. The occupancy of resources in the tuple is associated to an operating frequency. Multiple tuples for different operating frequencies can be assigned to a workload. Each task t_i consumes a fixed amount of each resource, r_{ik} that does *not* vary with time. *Slot* takes scheduling decisions for groups of tasks that we call a *Slot*. A slot s is defined by the tuple $(G_s, h_s, r_{s1}, r_{s2}, ..., r_{sk})$. $G_s \subseteq G$ is the graph of tasks associated to the slot and h_s is the Hardware Execution Time (HET) of the slot. The generic tuple $(r_{s1}, r_{s2}, ..., r_{sk})$ denotes the slot occupancy for each of the k resources (e.g., number of logic elements, memory, DSP blocks). Resources occupied by a slot correspond to the sum of the resources occupied by its constituent tasks. Obviously, the amount of resources of a slot cannot be larger than those available in the target FPGA: $r_{s1} = \sum_{t_i \in G_s} r_{i1} \leq R_1,\ r_{s2} = \sum_{t_i \in G_s} r_{i2} \leq R_2,\ ...,\ r_{sk} = \sum_{t_i \in G_s} r_{ik} \leq R_k$. Slots are executed sequentially, tasks within a slot cannot execute until all tasks in preceding slots have terminated. Slots are interposed by FPGA reconfigurations that add a latency denoted by T_R.

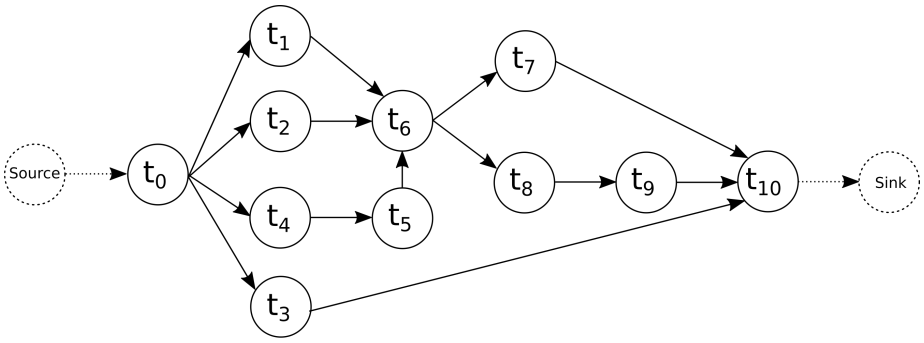


Figure 4.10: 13-tasks application DAG, used as case study for this Chapter

In **synthesis**, *Slot* heuristic iteratively transforms a DAG that expresses multiple partial execution orders into a DAG that expresses a single total execution order (a schedule). This is performed, at each iteration, by creating a *Slot*, thanks to the concept of *computational dominance*. A slot is built around the task that has the largest HET (dominating task) among unscheduled tasks. Dominated tasks are added to a slot, as long as there are enough FPGA resources, in a way that reduces the parallelism for further slots the least possible. After the generation of the first slot, the process is repeated by taking the task larger in timing among unscheduled. The final schedule is a succession of FPGA configurations, whose latency is determined by the sum of the dominating tasks (that hide the latencies of the dominated tasks).
To better explain *Slot* algorithm in Section 4.6.1, we will use an example. Specifically, we will show how the input DAG will be transformed, step-by-step, by *Slot*. Topology of the example is shown in Figure 4.10, whereas resources requests are expressed in Table 4.2. Values refers to a real FPGA, specifically Xilinx Virtex Ultrascale 9P FPGA [16], which is integrated on servers of Amazon Elastic Compute Cloud. According to the 3D resource model adopted in this Chapter, resource availabilities of such FPGA are listed in Table 4.3.

4.6.1 Pseudocode

Algorithm 1 shows the **pseudo-code** of *Slot*. Its core is constituted by a loop, lines 5-14, that iterates over a worklist where tasks are sorted in decreasing order of their HET (lines 3-4). From the motivating example in Section 1.4, we noticed that tasks with the highest HET impact more the total application makespan. So, regardless of its location within the DAG, we select the largest task in terms of execution timing from the worklist and we start to take scheduling decision from it (line 5). We name it *dominating task* t_i. In the first iteration of *Slot*, the dominating task of the Example in Figure 4.10 if t_3. Starting from the dominating task t_i, function $buildCandidateSlots()$ (line 7) computes a set S of *candidate slots*. For the sake of simplicity, we provide

Task	LEs [u]	DSPs [u]	EMBs [Mb]	HET [ms]
t_0	487k	1966	7	287
t_1	402k	2565	7	139
t_2	272k	1539	15	209
t_3	353k	1966	12	460
t_4	609k	513	18	200
t_5	704k	428	15	314
t_6	943k	1453	10	199
t_7	788k	1881	5	303
t_8	566k	428	5	35
t_9	1004k	599	9	49
t_{10}	291k	855	20	114

Table 4.2: The resource occupancy and HET of the tasks in Figure 4.10

LEs [u]	DSPs [u]	EMBs [Mb]	T_R time [ms]
2586k	6840	76	200

Table 4.3: Xilinx Virtex Ultrascale 9P FPGA - 3D Modelling

Function `generateSlots(` $G = < T,\ E >$ `):`
  $G' := G$;`/* Copy G to G', G'=<T',E'> */`
  $worklist \leftarrow T' \setminus \{t_{source}, t_{sink}\}$;
  $worklist \leftarrow sortInDecreasingOrderOfHET(worklist)$;
  foreach $t_i \in worklist$ **do**
    $S \leftarrow \emptyset$;`/* set of candidate slots */`
    $S \leftarrow buildCandidateSlots(t_i, G', S, R_1, R_2, ..., R_k, HET_{ti})$;
    foreach $s \in S$ **do**
      $scores[\ s\ ] \leftarrow computeScore(s, G')$;
    end
    $s \leftarrow retrieveLowestScoreSlot(scores[])$;
    $G' \leftarrow contractSubgraph(ss, G')$;
    $worklist \leftarrow worklist \setminus \{tasksins\}$;
  end
  $G'' \leftarrow minimizeReconfigurations(G')$;
return G";

Algorithm 1: *Slot* scheduling heuristic

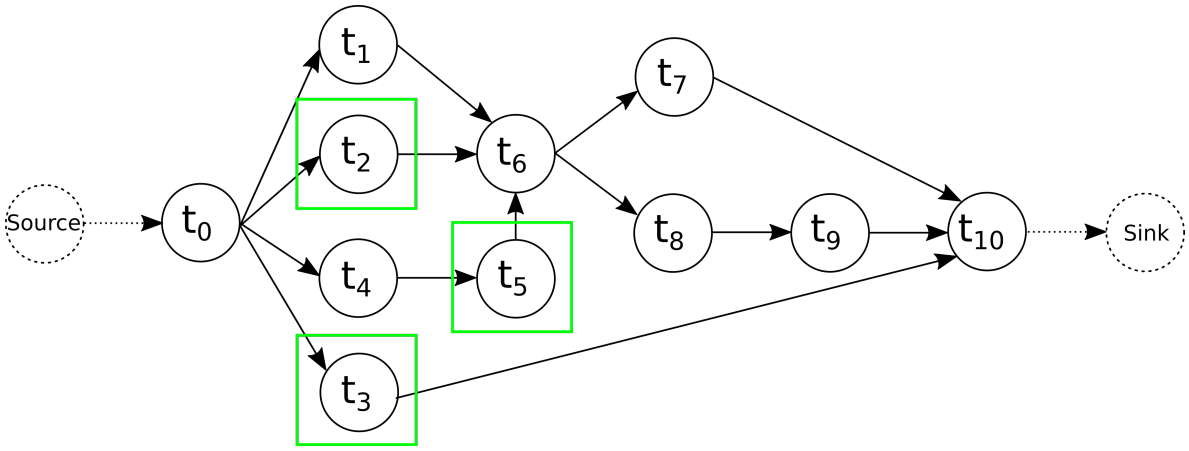


Figure 4.11: First slot computed by *Slot* with respect to the case study of Figure 4.10

here an intuitive description of its behavior (see Algorithm 2 and Section 4.6.2 for the details). A candidate slot is composed of a dominating task t_i and a *valid* set of dominated tasks. Such a set is composed of all combinations of tasks that can execute in parallel to t_i and fit the remaining FPGA resources (i.e., the total FPGA resources minus those occupied by t_i) and which respect the dependencies among tasks (i.e., a candidate slot cannot contain a task t_k and a task t_j, which is sequential to t_k, without all the tasks between t_k and t_j). Examples of valid candidates slots are those composed of tasks $\{t_3, t_2, t_5\}$, $\{t_3, t_2\}, \{t_3, t_7, t_9\}$. Examples of non-valid candidates slots are those composed of tasks $\{t_7, t_8, t_9, t_{10}\}$ (it occupies more resources than those offered by the FPGA) and $\{t_3, t_5, t_8\}$ (a candidate slot cannot contain tasks t_5 and t_8 without containing task t_6 too).

Among all candidate slots in S, only one is selected to be created in the current DAG G', lines 8-10 in Algorithm 1. This selection is based on the score returned by function $computeScore(s, G')$ that we detail in Algorithm 3 and Section 4.6.3. A score is an estimate of the makespan in the graph $G' - G_s$ that would result if we removed its tasks the subgraph of tasks created by slot s, G_s, from G'. Briefly, it tries to evaluates how much a candidate slot impacts the future makespan. At line 11, we select the slot with the lowest score, which is $\{t_3, t_2, t_5\}$ with respect to our example (see Figure 4.11).

This is the candidate slot for which the estimated makespan in $G' - G_s$ is the lowest. Therefore, creating this slot leaves the (estimated) highest degree of time-weighted parallelism in the residual DAG $G' - G_s$. Creating a slot is performed by contracting the nodes for the tasks of slot G_s into a single node, in G', by function $contractSubgraph()$ (line 12). The latter modifies G' by relabeling nodes that belong to G_s with the new slot identifier. It collapses the newly relabeled nodes by removing internal edges as well as duplicate cross edges (edges with an endpoint in the slot

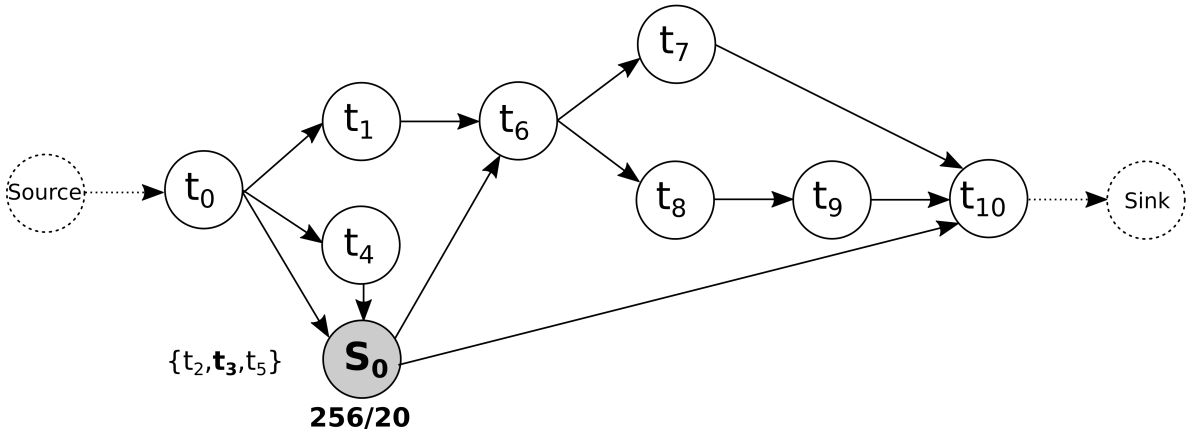


Figure 4.12: Merging of the tasks which compose the slot identified in Figure 4.11

and one in $G' - G_s$) and self-loops (edges whose endpoints are identical). With respect to the example, contraction of slot composed of tasks $\{t_3, t_2, t_5\}$ would result in slot S_0, as shown in Figure 4.12.

Once the winning slot has been selected and the DAG is transformed, tasks of G_s are removed by the worklist (line 13). The algorithm is repeated until there are tasks in the worklist, i.e. there are tasks to be scheduled. Figure 4.13 illustrates all the graph transformations that the heuristic performs from a partially order DAG of tasks (Figure 4.13a) to a totally ordered DAG of slots (Figure 4.13g). Each transformation corresponds to an iteration of the for-loop in Algorithm 1. Such a totally ordered DAG of slots constitutes a schedule.

The final optimization phase given in line 15 attempts to further improve the solution by removing useless reconfigurations. We will define this step in details in Algorithm 4 and Section 4.6.4.

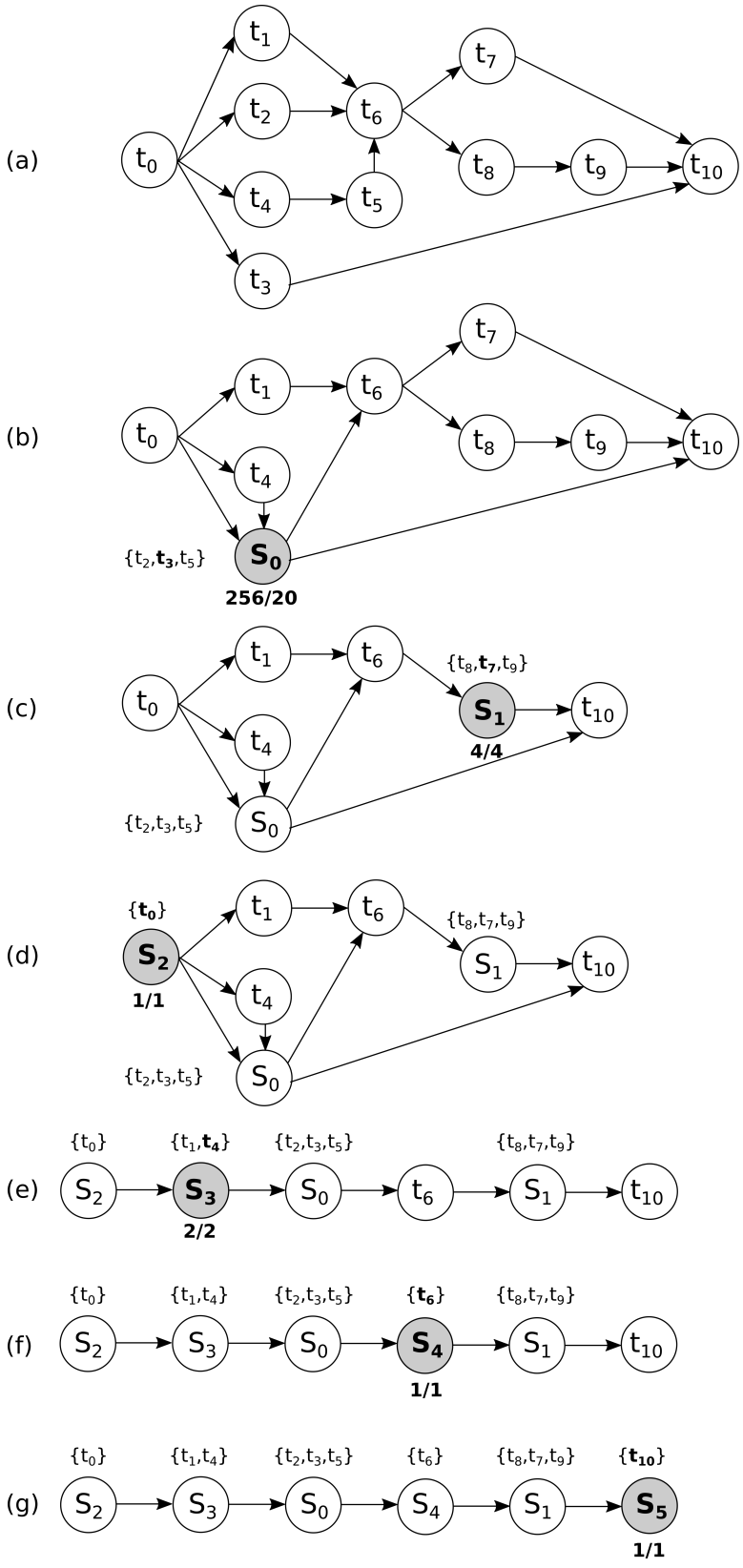


Figure 4.13: All the steps of *Slot* applied to an example. To keep the figure ligher, artificial source and sink are not represented

4.6.2 Candidates Generation

The pseudo-code for the creation of candidate slots for a dominating task t_i, function $buildCandidateSlots()$ (line 7 in Algorithm 1) is presented in Algorithm 2.

```
1  Function buildCandidateSlots( t_i, G' = < T', E' >, S, R_1, R_2, ..., R_k):
2      C ← G' \ {t_i, pred(t_i, G'), succ(t_i, G')};
3      foreach C' ∈ combinationsOfParallelTasks(C) do
4          G_s = {t_i};
5          s ← ({t_i}, h_i, r_i1, r_i2, ..., r_ik);
6          valid = true;
7          foreach t_c ∈ C' do
8              if executionTime(G_s + t_c, G') ≤ h_i then
9                  if validAllocation(s, t_c, R_1, R_2, ..., R_k) then
10                     s ←
                         (G_s+t_c, executionTime(G_s+t_c, G'), r_s1+r_c1, r_s2+r_c2, ..., r_sk+r_ck);
11                     continue;
12                 end
13             end
14             valid = false;
15         end
16         if valid == true then
17             S ← S ∪ {s};
18         end
19     end
20 return S
```

Algorithm 2: The function that builds the candidate slots.

Candidate slots are computed from C : a subgraph of the current DAG G', where the dominating task t_i, its successors and predecessors are removed. For instance, with respect to Figure 4.11, graph C is equal to the graph in figure without tasks t_0 and t_{10}, because they are respectively predecessor and successor of the dominating task t_3. For the sake of precision, we specify that functions $pred(t_i, G')$ and $succ(t_i, G')$, line 2, return the set of predecessors (from the source) and successors (up to the sink) of a task $t_i \in G'$, respectively. Successors and predecessors are removed because *Slot* algorithm is based on the concept to paralellize the best subsets of unscheduled tasks with the dominating task, namely the task among unscheduled which very likely will have a strong impact on the overall makespan. Thus, at this step of the algorithm, the dominating task is not sequentialized to any other task. Function $combinationsOfParallelTasks()$, line 3 in Algorithm 2, returns the c-combinations of tasks in the subgraph $K \subseteq G'$, with $c = 1, ...|C|$ that can be executed in parallel to a dominating task. In Figure 4.11, for the dominating task t_3, this

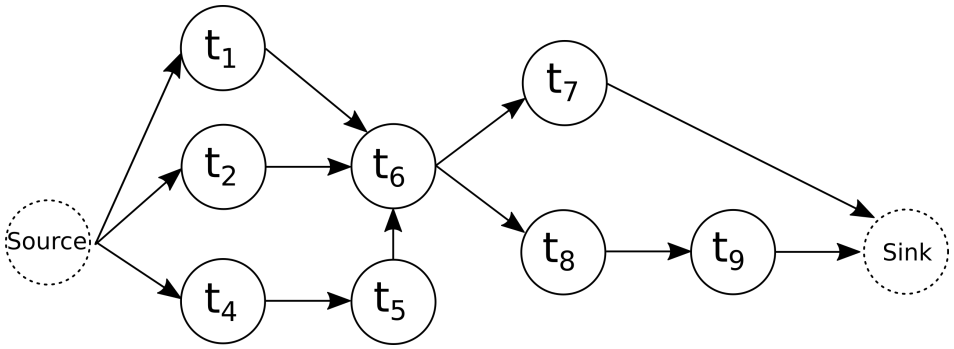


Figure 4.14: Removing tasks t_0, t_3 and t_{10} from case study of Figure 4.10

function returns the combinations of $c = 1, 2, ..., 8$ tasks that can execute in parallel to t_3, from the subgraph obtained by removing t_0, t_3 and t_{10}, depicted in Figure 4.14. Candidates are generate by adding one task at a time (line 7). However, some of these combinations are invalid and must be filtered out (lines 8-13 in Algorithm 2). Valid combinations are those whose tasks respect three conditions:

1. **Resources-availability condition:** the sum of scheduling-independent resources, which are the only ones taken into account in this chapter, shall not exceed those offered by the FPGA. Function $validAllocation()$, line 9, verifies if a slot disposes of enough FPGA resources to accomodate at least one remaining task.

2. **Topology condition:** constraints on topology may affect the generation of candidates in a dual way.

   a) Constraints among unscheduled tasks - as we already seen in Section 4.6.1, a candidate slot cannot contain a task t_k and a task t_j, which is sequential to t_k, without all the tasks between t_k and t_j. This is the case for the candidate $\{t_3, t_5, t_8\}$ in Figure 4.14. Such candidate contains tasks t_5 and t_8, but not task t_6 and then it is topologically non-valid.

   b) Constraints between unscheduled tasks and already formed slots - the same principle as above can be applied to already formed slots. Figure 4.14 does not provide us a valid example because it refers to the first iteration of Algorithm 1 and therefore no slots are still formed. Let us consider Figure 4.13b instead. Regardless the scheduling choices taken in the rest of Figure 4.13b, a candidate which contains both tasks t_4 and t_6 would be topologically non-valid. Indeed, task t_4 precedes slot S_0 which in turn precedes task t_6. Since slots must be executed in sequence, any situation in which an already defined slot (such as S_0) is placed between

a set of tasks (such as t_4 and t_6) makes topologically non-valid scheduling such set of tasks in the same slot.

3. **Computational-dominance condition:** the total HET required by all tasks of a candidate slot cannot exceed the HET of the dominating task for such slot. In other words, the execution of all the tasks which are parallel to the dominating task (i.e., dominated tasks), is hidden by the execution of the dominating task. This implies the fact that the HET of the dominated tasks has no impact with respect to the overall makespan. The choice of which tasks are dominated in which slot plays a key-role in the minimization of the makespan. Function $executionTime(X, G')$, line 8, is used to verify if a combination of tasks X respects the computational dominance principle in the DAG G'. It returns the length of the critical path that tasks in X form in G'. For $X = \{t_2, t_4, t_5\}$ in Figure 4.14, the function returns $max\{HET_2, (HET_4 + HET_5)\}$, which does not respect the computational-dominance condition (indeed, $HET_4 + HET_5$ = 514 ms and this exceeds the HET of the dominating task t_3, which is 460 ms).

With respect to the example, for dominating task t_3 we calculated 256 candidates, but only 20 of them did pass all the feasibility tests, as labeled in Figure 4.13, below each slot. Back to Algorithm 2, operation $G_s + t_c$, at line 10, adds t_c to the slot task graph G_s. This addition produces the same graph as the subtraction $G - G_s - t_c$. If the generated candidate s respects all the feasibility constraints, it is added to the set of valid candidates S (lines 16-18), which is returned at the end of the algorithm (line 20), after all the combinations of tasks which are parallel to the dominating one have been explored (loop in lines 3-19).

4.6.3 Score

Once a list of valid candidate slots is provided from function $buildCandidateSlots()$ (line 7 in Algorithm 1), it is necessary to evaluate which one among them is expected to lead to the best overall scheduling. When targeting reconfigurable hardware, a schedule is constrained by two elements. Thus, score is computed by separately considering inter-task dependencies and HETs, from the reconfiguration time T_R and the occupancy of tasks resources. Hence, the two terms returned by Algo-

1 **Function** `computeScore(` *slot* s, *dependency graph* G' `)`**:**
2 $J := merging\ G_s\ in\ G'_{copy}$; $\bar{r}_{J1} := \frac{\sum_{t_i \in J} r_{i1}}{R_1}$; $\bar{r}_{J2} := \frac{\sum_{t_i \in J} r_{i2}}{R_2}$; ...; $\bar{r}_{Jk} := \frac{\sum_{t_i \in J} r_{ik}}{R_k}$;
3 $n_J^{reconfig} := max(\lceil \bar{r}_{J1} \rceil, \lceil \bar{r}_{J2} \rceil, ..., \lceil \bar{r}_{Jk} \rceil)$;
4 **return** $T_\infty(J) + n_J^{reconfig} \times T_R$;

Algorithm 3: The function that assigns a score to a slot

rithm 3. The first term, $T_\infty(J)$ quantifies the impact of tasks HET and inter-task dependencies (that impose scheduling constraints) in graph J, the graph obtained by merging tasks of candidate slot G_s in a copy of graph G'. Differently from function $contractSubgraph(G_s, G')$, line 12 of Algorithm 2, $computeScore(s, G')$ (detailed in Algorithm 3) does not actually modify nor overwrite G', because it acts on a copy of it. Calculating $T_\infty(J)$ is equivalent to calculating the makespan of unscheduled tasks in DAG J, by scheduling them onto an FPGA characterized by infinite resources. In this manner, we evaluate the level of HET-weighted parallelism among tasks which remain in graph G' if we would choose slot G_s as a winning slot. $T_\infty(J)$ considers only the impact of HETs and data-dependencies by ignoring resources occupancy. The latter has a contribution on the overall makespan, because the estimated number of reconfigurations in graph J, multiplied per the reconfiguration time T_R, determines the estimated time spent in future reconfigurations. This is taken into account by the second parameter of the score, which calculates the minimum number of reconfigurations in graph J by considering the occupancy of the tasks only without consider inter-task dependencies. For each scheduling-independent resource (i.e., LEs, DSPs, EMBs in this example) r_i, we calculate the sum of requests of tasks which belong to graph J, $\sum_{t_i \in J} r_{i1}$. This sum is divided by the total availability of the FPGA for such resource R_i and the next integer (provided by ceil ($ceil\rceil$) function), named $\bar{r}_i$, is taken. The overall minimum number of reconfigurations is given by the largest among each $\bar{r}_i$. We chose to optimistically consider the *minimum* number of reconfigurations because, from our simulations, the number of slots of the optimal solutions was often equal to the number of slots calculated through Algorithm 3. Such value, multiplied per T_R gives the time contribution given by future reconfigurations.

As an example, let us calculate the score for the slot $\{t_3, t_2, t_5\}$. Graph G' (equivalent to the DAG depicted in Figure 4.13a) is duplicated in graph J and tasks $\{t_3, t_2, t_5\}$ are merged onto slot S_X in graph J by obtaining a situation illustrated in Figure 4.15. We calculate the two contributions over such graph. In this respect, neither S_X nor other already defined slots (not present in Figure 4.15 because it refers to the first iteration of Algorithm 1) contribute on the two parameters. In other words, all the slots (which can be already defined or under evaluation such as S_X) have a null contribution over the calculus of $T_\infty(J)$ or $n_J^{reconfig} \times T_R$. This is because we are only interested on the impact derived by scheduling slot S_X onto tasks which are still unscheduled. $T_\infty(J)$ is given by: HET_{t_0} + $max(HET_{t_4}, HET_{t_1})$ + HET_{t_6} + $max(HET_{t_7}, HET_{t_8+t_9})$ + HET_{t_10} = 287 + 200 + 199 + 303 + 114 [ms] = 1103 [ms]. The minimum number of reconfigurations is given by: $n_J^{reconfig} := max(\lceil \bar{r}_{LEs} \rceil, \lceil \bar{r}_{DSPs} \rceil, \lceil \bar{r}_{EMBs} \rceil)$, where:

$\lceil \bar{r}_{LEs} \rceil$ is given by the ceil of the sum of LEs required by tasks in J divided by availability of LEs of FPGA (2586k in this example), namely: $\lceil 5090k/2586k \rceil$ = $\lceil 1.968 \rceil$ = 2.

$\lceil \bar{r}_{DSPs} \rceil$ is given by the ceil of the sum of DSPs required by tasks in J divided by avail-

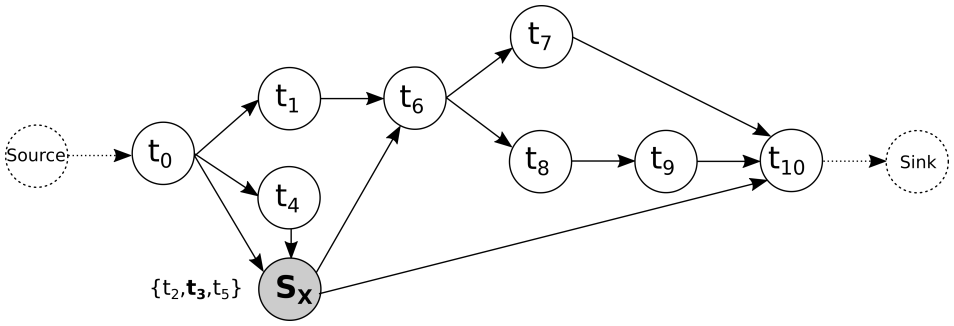


Figure 4.15: DAG considered by the scoring-based system for the first iteration of *Slot*

ability of DSPs of FPGA (6840 in this example), namely: $\lceil 10266/6840 \rceil$ = $\lceil 1.5 \rceil$ = 2.

$\lceil \bar{r}_{EMBs} \rceil$ is given by the ceil of the sum of EMBs Mb required by tasks in J divided by availability of EMBs Mb of FPGA (75.9 in this example), namely: $\lceil 81/75.9 \rceil$ = $\lceil 1.06 \rceil$ = 2.

The estimated time which will be passed by tasks of G' by choosing tasks $\{t_3, t_2, t_5\}$ as a winning slot is: $2 * T_R$. The latter parameter is fixed to $200ms$ in this example, for a total of $400ms$.
The score for slot $\{t_3, t_2, t_5\}$ is then: 1103 + 400 ms = 1503 ms. This number can be read as follow: *by scheduling tasks* $\{t_3, t_2, t_5\}$ *together, remaining tasks in graph* G' *would need at least total ms to be executed*. In this respect, the actual number calculated once the scheduling has been defined is 1477 ms. As the iterations go on, such estimation tends to become more precise, because both the number of unscheduled tasks and the complexity decrease. In order to take a scheduling decision, we do not need to calculate a perfect estimation, but only that the calculated numbers respect the real quality of each candidate. In this sense, this score experimentally demonstrated to work well (refer to evaluation, in Chapter 5).

We highlight that such score allows the heuristic to take scheduling decisions that a user normally considers counter-intuitive. For instance, slot composed of tasks $\{t_2, t_3, t_5\}$ is preferred over 19 other resource-valid candidates, such as $\{t_2, t_3, t_4\}$. At a first glance, a slot composed of tasks $\{t_2, t_3, t_4\}$ may seem more effective, because all the tasks are parallel and they belong to the same breadth-first search level (i.e., they are all far 2 steps from the source of the graph). In this respect, score understood that slot $\{t_2, t_3, t_5\}$ amortizes the execution time of t_5 (larger than that of t_4) as t_5 can execute in parallel to the dominating task t_3. In addition, score evaluated that, even though t_5 is one breadth-first search level far more from the source rather than t_3 and t_2, slot $\{t_2, t_3, t_5\}$ is still preferable over the other 19 candidates. This choice is

not trivial because it depends on a combination of parameters: data-dependencies, HETs of tasks, reconfiguration time, number of reconfigurations, and so on. With respect to this example, in fact the optimal solution provides slot $\{t_2, t_3, t_5\}$ and it is the same as the one calculated by *Slot*. Not limited to this example, we experimentally demonstrated in Chapter 5, that the scoring system often led to select a good choice.

4.6.4 Optimization Phase

As illustrated in Figure 4.13, we compute a schedule by progressively transforming an initial tasks DAG, which defines a partial order for tasks, Figure 4.13a, into a slot DAG that specifies a total execution order for both slots and tasks, Figure 4.13g.

While designing the heuristic, we observed that during the final iterations of Algorithm 1, slots tend to be composed of a single dominating task t_i, see Figure 4.13e, Figure 4.13f and Figure 4.13g. This is because most of the candidate dominated tasks have already been assigned to slots in previous iterations. As an additional improvement of the results, in order to improve the FPGA utilization and saving more reconfigurations, we propose Algorithm 4. Here, we scan all slots in the slot DAG and, for each slot, we attempt to allocate its tasks to a neighboring slot, in a first-fit manner. This re-allocation is performed by means of contracting edges between slots. Edge contraction is defined in [57] as the operation that removes an edge from a graph, while merging the edge's end vertices and removing duplicate edges. Tasks of a slot are allocated to the first neighboring slot s' that has enough FPGA resources and for which dependencies are respected. All tasks in s' must either be predecessors or successors of t_i in the initial DAG G. Figure 4.16 illustrates how function

```
1  G' ← generateSlots(G);
2  G' ← reduceReconfigurations(G');
3
4  Function reduceReconfigurations( slot DAG G' =< S, L > ):
       /* S := set of slots, L := set of slot arcs                    */
5      foreach s ∈ S do
6          foreach s' ∈ {S \ s} | ∀t_i ∈ T_s', t_i ∈ pred(s, G') ∨ t_i ∈ succ(s, G') do
7              if (r_s'1 + r_s1 < R_1) ∧ (r_s'2 + r_s2 < R_2) ∧ ... ∧ (r_s'k + r_sk < R_k) then
8                  contractEdge(s → s', G');
9                  break;
10             end
11         end
12     end
13 return G';
```

Algorithm 4: Merging single-task slots in first-fit.

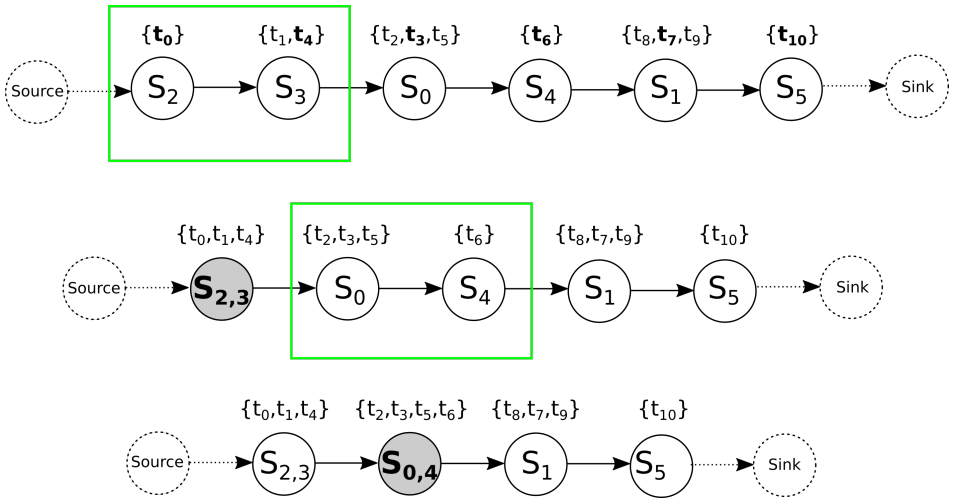


Figure 4.16: Reducing the makespan of Figure 4.13g as described in Algorithm 4.

$reduceReconfigurations()$ reduces the latency for the slot DAG of Figure 4.13g. It merges S_2 and S_3 in the new slot $S_{2,3}$ and it merges S_0 and S_4 in the new slot $S_{0,4}$. The improved DAG in Figure 4.16c contains 4 slots (instead of 6 in Figure 4.13g) by saving $2 * T_R = 400ms$. Algorithm 4 is simple yet efficient enough to produce solutions that are very close to the optimum (see Chapter 5). In fact, the problem solved by Algorithm 4 is a research problem which is simpler than the one solved by *Slot*, for which a heuristic is necessary in order to keep the run-time low. Thus, we cannot exclude that exact solution (or almost exact solutions) which solve the slots compacting problem by keeping the run-time limited to the order of tens of milliseconds may exist. Based on our experiments, an algorithm simple and fast such as Algorithm 4 (its complexity is linear with the number of slots) was enough to produce results which are close-to-optimum.

4.6.5 Complexity discussion

For each component of *Slot* we provide a theoretical complexity, which corresponds to the worst-case, and a practical complexity, which corresponds to the average case.

Table 4.4 illustrates our considerations about complexity of *Slot* algorithm. We distinguish between four main contributions (i.e., the contributions which are computationally heaviest): the main loop of Algorithm 1 (i.e., lines 5-14), function $buildCandidateSlots$ (line 7 of Algorithm 1 and further detailed in Algorithm 2), function $computeScore$ (line 9 of Algorithm 1 and further detailed in Algorithm 3) and function $minimizeReconfigurations$ (line 15 of Algorithm 1 and further detailed in Algorithm 4).

The main loop of Algorithm 1 iterates over a worklist composed of the sequence of tasks that compose the input DAG ordered for decreasing HET. Thus, iteration in

Algorithm	Theoretical complexity	Practical complexity
Main loop of Alg. 1	*O*(|N|)	*O*(|N|)
Candidate gener. - Alg. 2	*exponential**	*polynomial**
Score - Alg. 3	*O*(|N| + |E|)	*O*(|N| + |E|)
Optimization - Alg. 4	*O*(|slots|)	*O*(|slots|)

Table 4.4: Computational complexity of the main steps of *Slot*. We remind the reader that $|N|$ represents the number of nodes of the application DAG (i.e., the number of tasks) and $|E|$ represents the number of edges (i.e., the data-dependencies between tasks)

lines 5-14 of Algorithm 1 is repeated a number of times that is equal to the number of tasks in the worst-case. This is equivalent to the situation in which each task composes a single-task slot. In the average case, the main loop is repeated a number of times which is under the number of tasks in the DAG (i.e., some tasks are aggregated). We left the analysis of function $buildCandidateSlots$ and Algorithm 2 for the end of this section and we pass to the analysis of function $computeScore$ and Algorithm 3. In our implementation, we calculate both contributions of the score through a single breadth-first visit of the graph. In the worst case, every vertex and every edge will be explored. *O*(|E|) may vary between *O*(1) and *O*(|N|^2), depending on how sparse the DAG is [38]. As the iterations go on, the original DAG is simplified because of the effect of function $contractSubgraph$, line 12 of Algorithm 1, making the breadth-first visit less complex. Our implementation of $minimizeReconfigurations$ function (Algorithm 4) has a complexity linear with the number of nodes in the slot DAGs (e.g., DAG in Figure 4.13g).

Thus, the real **complexity** of the heuristic is determined by the number of combinations of tasks that may form a slot, for the subgraph C defined at line 2 in Algorithm 2. This number depends on the task dependencies in C and cannot be expressed in closed form. In the worst case, for a graph C where all tasks can execute in parallel, the number of combinations amounts to $\sum_{i=1}^{|N_C|} \binom{|N_C|}{i}$, where $|N_C|$ is the number of tasks in C[1]. Fortunately, these highly parallel graphs are almost never encountered in practice. In fact, the total number of combinations is strongly limited by task dependencies and by resource constraints. In most of the practical cases we encountered, the complexity is maximal at the first iterations of the loop at line 3 in Algorithm 2. Complexity decreases significantly with the creation of subsequent slots as parallelism in G' is progressively reduced. This can be seen in Figure 4.13 where below each slot we reported a pair of numbers f/g. f is the number of combinations

[1]This also corresponds to the theoretic case of a graph with no edges (null graph). We ignore this case as it violates our design assumptions.

in C that can be computed without considering for inter-task dependencies (the theoretical complexity). g is the number of *valid* candidate slots (the actual complexity). A significant difference between f and g exists only for S_0. In our implementation, we combined function $combinationsOfParalleTasks()$ with the tests at lines 7 and 10. When a combination of tasks X does not respect the computational dominance condition or requires more FPGA resources than those available, we stop exploring combinations that are descendants of X. This prunes the candidate space and significantly reduces run-time. Resuming, the generation of candidate slots surely constitute the most expensive operation of *Slot* from time complexity point of view. Even though the theoretical complexity of this function is exponential, this is not not reflected in practical cases for several reasons. Firstly, data-dependencies and constraints that defines the feasibility of a candidate slot reduce the number of candidate slots. Secondly, as the iterations go on, more the input DAG is simplified because merging tasks together reduce the parallelism of the graph. If we deal with large application graphs (e.g., hundreds of tasks), some additional heuristics which avoid to consider all the combinations of tasks which are parallel to a dominating tasks can be considered. Examples in such sense are (i) considering only slots which maximize the resource occupancy of the FPGA (in other words, that means to discard all the slots whose tasks are entirely contained in another valid slot) and (ii) forbid to place in parallel tasks whose breadth-first distance from the source of the graph exceeds a certain threshold. However, these are only ideas since we did not move our research in such sense.

Precisely estimating the timing complexity of this part of *Slot* algorithm is a complex problem which we did not entirely taken into account in this thesis. However, we did some experimental considerations in this respect. In the face of a theoretical complexity which is exponential, in the benchmark presented in Chapter 5 we noticed a practical complexity which tends to be polynomial with the number of tasks in the input DAG.

4.7 Conclusion

In this chapter we have described how our approach, named *Slot*, can efficiently tackle the FPGA scheduling problem. At the beginning of the chapter, we have shown why an exhaustive analysis is unfeasible from the computational timing point of view. Then, we have described how we capture hardware resources and applications. Finally, we have described the main steps of *Slot* and we have discussed its theoretical complexity. We think that the key-strenght of *Slot* is its intelligent computation of slots, which efficiently exploit the parallelism of an FPGA. Next chapter will show the evaluation of *Slot* on a synthetic benchmark composed of several pseudo-randomly generated instances. Specifically, we will evaluate the quality of

the scheduling solution produced by *Slot* and its run-time.

Chapter 5

Experimental evaluation

5.1 Introduction

This chapter experimentally evaluates *Slot* and compares it with other, exact or approximate, approaches.

In order to evaluate the quality of the schedule computed by *Slot* we must compare the obtained makespans with an optimal absolute reference. We thus first formally model the FPGA scheduling problem and use this to solve problem instances with an exact approach. We chose to use Mixed Integer Linear Programming (MILP) because it seams very suitable for this kind of optimization problems. We then design an instance random generator to generate a large number of instances of the FPGA scheduling problem. We solve these instances with an MILP solver to obtain the minimal makespans and compare them with the makespans found by *Slot* on the same instances. We also compare the run-times of the two approaches. Finally, we compare the quality and the run-times of *Slot* with the HEFT-NF heuristic that we already discussed in Chapter 3. To the best of our knowledge HEFT-NF is the only proposal comparable to *Slot*.

Section 5.2 presents our formal description suitable for MILP solving. Section 5.3 describes the instances generator we used to generate about 37500 instances of the FPGA scheduling problem. Section 5.4 shows the results obtained by MILP, HEFT-NF and *Slot* on this benchmark and discusses them. Section 5.5 concludes the chapter.

5.2 MILP Formulation

Until now we discussed the FPGA scheduling problem based on an informal description. A formal model is needed to feed an exact solver and obtain exact optimal solutions for instances of the FPGA scheduling problem. The choice of a formu-

lation suitable for MILP solving comes from the characteristics of the problem: its constraints are time-related (dependent tasks must execute in a correct order) and resources-related (the FPGA available resources shall never be exceeded). These constraints can conveniently be expressed using numeric variables (start and end times, amounts of available or consumed resources) and inequalities. On top of this the optimization goal is easily expressed as a simple objective function: the makespan. For all these reasons MILP is a good candidate exact solver. As we do not know what the theoretical complexity of the problem is, we do not know how an MILP solver will perform and what is the maximum instance size it can solve in a reasonable amount of time but this is one of the expected outcomes of this experiment: get an estimate of the practical complexity.

Each problem instance is entirely specified by a set of input values, a set of constraints and an objective function[1]. Input values can be, for instance, the number of tasks or their durations. From these input values, the constraints and the objective function, the MILP solver computes output variables like the start times of tasks or the task-to-slot allocations.

Inputs

The input values used in our formulation are:

- $n_t \in \mathbb{N},\ n_t \geq 2$ is the total number of tasks, including the artificial source and sink. In the following we denote $\mathfrak{T} = \{T_0, \ldots, T_{n_t-1}\}$ the set of tasks. T_0 and T_{n_t-1} are the artificial source and sink tasks with zero duration and zero resource consumption. We remind that the artificial source and sink tasks are just a way to close the DAG that represents the inter-task dependencies and that they are added without loss of generality. They ease the modelling by providing simple start and end points but they have no impact on the results.

- $\boldsymbol{y} \in \mathbb{R}^{+n_t}$ is the vector of the durations of the tasks; y_t is the duration of task T_t. We always have $y_0 = y_{n_t-1} = 0$ because tasks T_0 and T_{n_t-1} have zero duration.

- $n_r \in \mathbb{N}, n_r \geq 1$ is the number of types of resources offered by the target FPGA. $\mathfrak{R} = \{R_1, \ldots, R_{n_r}\}$ is the set of types of resources (e.g. LE, DSP blocks, Embedded Memory Blocks, Clock Generators...)

- $n_s = n_t - 2 \in \mathbb{N}$ is the total number of slots. $\mathfrak{S} = \{S_1, \ldots, S_{n_s}\}$ is the set of slots. By definition the maximum number of slots is equal to the number of non-artificial tasks so, to simplify the problem specification, we fix the number of slots to its theoretical maximum ($n_s = n_t - 2$). In most cases the last slots of a schedule have no allocated tasks and are not considered in the computed

[1] Plus the semantics of the model, of course

makespan, but in exceptional cases it can be that all slots are used, each by one single task.

- $\mathcal{T} \in \mathbb{R}^+$ is the reconfiguration time of the target FPGA.

- $P \in \{0,1\}^{n_t \times n_t}$ is a matrix of precedence relations; elements of P are binary values; if P element $\pi_{t,t'} = 1$, then task T_t precedes task $T_{t'}$ ($T_{t'}$ can execute only after T_t termination). Precedence is transitive so all precedence matrices with same transitive closure are equivalent. P is not a free variable and must obey constraints: the directed graph G with vertices in $\mathfrak{T}$ and edges defined by P must be acyclic. P must also be such that task T_0 is a direct or indirect predecessor of all other tasks and task T_{n_t-1} is a direct or indirect successor of all other tasks.

- $\boldsymbol{q} \in \mathbb{N}^{n_r}$ is the vector of the total available quantities of FPGA resources; q_r is the total quantity of resource R_r. We decided to use natural number values because FPGA hardware resources are usually discrete elements. As these are input variables, and no output integer variables are derived from them, this choice has no impact on the solving complexity. If other types of resources were needed and would be better described by real numbers (e.g. energy), this choice could easily be changed and would have no impact.

- $R \in \mathbb{N}^{n_t \times n_r}$ is the matrix of resources consumptions; elements of R are natural numbers; R element $0 \leq \rho_{t,r} \leq q_r$ is the consumption of resource R_r by task T_t (we consider only problem instances for which solutions exist, so there are no tasks that consume alone more resources than what is available). The artificial source and sink tasks do not consume any resource:

$$\forall\ 1 \leq r \leq n_r\ \rho_{0,r} = \rho_{n_t-1,r} = 0$$

- $\mathcal{H}$ is a large number, larger than every possible makespan. In MILP parlance this is the "horizon". In our case $\mathcal{H}$ can easily be set to the sum of the task durations plus the sum of the maximum number of reconfiguration times. This extreme case corresponds to the worst possible schedule where all slots are used with one single non-artificial task per slot:

$$\mathcal{H} = n_s \times \mathcal{T} + \sum_{0 \leq t < n_t} y_t$$

Outputs

The outputs are the values computed by the MILP solver and that fully define a solution. They must be very carefully selected and their types (real, integer, binary) must

also be carefully chosen because these choices have a strong impact on the solving complexity. The theoretical complexity of the general Linear Programming (LP) problem is polynomial and instances can actually be solved in polynomial time using interior-point techniques. Changing some of its real output variables into integer output variables changes the LP problem into MILP and the theoretical complexity becomes NP-hard. The rule of thumb of our formulation is thus to limit the number of integer output variables and use binary variables instead of integer variables whenever possible.

- $\boldsymbol{x} \in \mathbb{R}^{+n_t}$ is the vector of start times of tasks; x_t is the start time of task T_t. Without loss of generality we constrain the start time of the artificial source task to be zero: $x_0 = 0$. The x_0 component is thus technically an input. Note that this means that the initial reconfiguration time is not counted in the total makespan, which is the same convention used by the two heuristics we compare with MILP. This convention is a bit more convenient on a pure implementations point of view and it does not change anything to the schedules selected by the three methods. It is just their claimed makespans that are systematically shorter by one reconfiguration time. The missing initial reconfiguration time is added to all makespans in a post-processing such that relative makespan comparisons are correct.
- $\boldsymbol{z} \in \mathbb{R}^{+n_t}$ is the vector of end times of tasks; z_t is the end time of task T_t. If we consider the x_t as true output variables, as task durations y_t are known inputs, the z_t are not true output variables: they can be computed from the x_t and the y_t and they should not add to the complexity.
- $\boldsymbol{u} \in \mathbb{R}^{+n_s}$ is the vector of start times of slots; u_s is the start time of slot S_s. Without loss of generality we constrain the start time of the first slot to be zero: $u_1 = 0$. The u_1 component is thus technically an input. As for x_0, this convention is a bit more convenient on a pure implementations point of view and does not change the computed schedules; the omitted initial reconfiguration time is added to the computed makespan in a post-processing.
- $\boldsymbol{v} \in \mathbb{R}^{+n_s}$ is the vector of durations of slots; v_s is the duration of slot S_s.
- $\boldsymbol{w} \in \mathbb{R}^{+n_s}$ is the vector of end times of slots; w_s is the end time of slot S_s. A bit like for the x_t, y_t and z_t, the u_s, v_s and w_s are redundant: the w_s, for instance, can be computed from the u_s and the v_s and they should not add to the complexity.
- $A \in \{0,1\}^{n_t \times n_s}$ is the allocation matrix; elements of A are binary values. A element $\alpha_{t,s} = 1$ if task T_t is allocated in slot S_s, else $\alpha_{t,s} = 0$. The $\alpha_{t,s}$ are the unavoidable but only source of MILP solving complexity. Without loss of generality we constrain task T_0 to be allocated to slot S_1: $\alpha_{0,1} = 1$ and $\forall\ 1 < s \leq n_s, \alpha_{0,s} = 0$. These matrix elements are thus technically inputs.

Objective function

Our objective function is the total makespan and can be very easily expressed using the output variables: it is $z_{n_t-1} - x_0 = z_{n_t-1}$, the end time of the artificial sink task T_{n_t-1}. The MILP solver will be instructed to find a solution that complies with the constraints and that minimizes z_{n_t-1}. This objective function does not involve the number of actually used slots, which naturally solves the potential issue about the total number of slots $n_s = n_t - 2$; the unused slots, if any, and the corresponding reconfiguration times are not counted in the objective function and do not influence the optimization effort.

Constraints

We present here a high-level human-readable form of the constraints of the MILP formulation. The low-level form that can directly be used by the solver is less intuitive. It makes use of auxiliary variables defined by linear equations of the input and output variables. A low-level constraint then consists in specifying upper and/or lower bounds of the output and auxiliary variable, plus type constraints on the output variables (e.g. the $\alpha_{t,s}$ are binary values).

For instance, in order to express that a task T_t cannot be pre-empted and its end time is its start time plus its duration we introduce the auxiliary variables a_t such that:

$$\forall\ 0 \leq t < n_t,\ a_t = x_t + y_t - z_t$$
$$\forall\ 0 \leq t < n_t,\ 0 \leq a_t \leq 0$$

We do not detail here this low-level form and the auxiliary variables. The reader interested in the low-level details of the MILP formulation will find them in Appendix A.

The list of time-related constraints is the following:

- A task cannot be pre-empted, its end time is equal to its start time plus its duration:

$$\forall\ 0 \leq t < n_t,\ z_t = x_t + y_t$$

- There is no idle time between slots and the start time of a slot is the end time of the previous slot plus the reconfiguration time:

$$\forall\ 2 \leq s \leq n_s,\ u_s = w_{s-1} + \mathcal{T}$$

- A slot cannot be pre-empted, its end time is equal to its start time plus its duration:

$$\forall\ 1 \leq s \leq n_s,\ w_s = u_s + v_s$$

- If a task T_t is allocated in a slot S_s ($\alpha_{t,s} = 1$), then its execution happens during the slot's duration:

$$\forall\ 0 \leq t < n_t,\ \forall\ 1 \leq s \leq n_s,\ \alpha_{t,s} = 1 \Rightarrow u_s \leq x_t \leq z_t \leq w_s$$

- If a task T_t precedes another task T'_t ($\pi_{t,t'} = 1$), then T_t end time is less or equal $T_{t'}$ start time:

$$\forall\ 0 \leq t < n_t - 1,\ \forall\ 1 \leq t' < n_t,\ \pi_{t,t'} = 1 \Rightarrow z_t \leq x_{t'}$$

The resources-related constraints can be condensed in one single statement: the tasks allocated to a slot cannot use more of any resource than its total available quantity:

$$\forall\ 1 \leq r \leq n_r,\ \forall\ 1 \leq s \leq n_s,\ \sum_{0 \leq t < n_t, \alpha_{t,s} = 1} \rho_{t,r} \leq q_r$$

Finally one more constraint is needed to express the fact that a task is allocated to one and only one slot:

$$\forall\ 0 \leq t < n_t,\ \exists!\ 1 \leq s \leq n_s, \alpha_{t,s} = 1$$

5.3 Random generation of problem instances

In order to build a large collection of instances with thousands of samples we could not use real world examples, first because we would never find such a large number of well documented designs, second because modelling them by hand according our MILP template would represent an enormous and error prone work. We thus decided to generate instances randomly.

An instance of the FPGA scheduling problem is characterized by its number of tasks, its DAG, the tasks durations and resources consumptions. Randomly assigning task durations or resources consumptions does not pose a problem, apart

deciding of bounds. In order to cover a large scope we eliminated too small resource consumptions because they tend to relax the resource-related constraints in favour of the time-related constraints and somehow bias towards a different, more classic, type of scheduling problems. Indeed, if all tasks fit in a single slot because they consume very few of each resource, the problem is not an FPGA scheduling problem any more. We also eliminated too large resource consumptions because they tend to over-simplify the problem: each task with a large consumption of one resource tends to become a slot by itself and cuts the rest of the schedule in two simpler halves. For our benchmark we decided that each task can consume any proportion between 10% and 50% of each available resource. Our experiments show that this produces many "hard" instances with non-trivial optimal solutions.

A similar reasoning led to our choice for the task durations. Too short tasks tend to bias the problem towards a resource-only optimization while too long tasks tend to favour our approach based on dominant tasks first. We decided that each task can last between 1/4 and 4 times the reconfiguration time of the FPGA. There again our experiments show that a lot of "hard" instances are generated.

The DAG random generation is a bit more difficult because of the acyclic constraint, the unique source and sink tasks and the need to obtain "realistic" dependency graphs that can be considered as representative of real-world applications. Indeed, the DAG of real world applications do not have arbitrary large number of edges; tasks with many producers and/or with many consumers are rare. We integrated all these constraints as follows:

- The DAG random generator is given a number of vertices $n_t > 2$ and a number of *internal* edges n_e. An internal edge is an edge that does not connect vertex T_0 or vertex T_{n_t-1}. The vertices represent tasks and the edges the direct dependencies between them.
- The task (vertex) indexes are used as a topological ordering of the dependency graph: if there is an edge from vertex T_i to vertex T_j, then $i < j$. This is a necessary and sufficient condition to guarantee that the graph is acyclic.
- The generator starts with a graph with n_t vertices and no edges. It iterates n_e times and adds one internal edge per iteration. At each iteration the generator randomly selects two different vertices T_i and T_j such that $0 < i < j < n_t - 1$ and there is no edge already between them. In order to avoid unrealistic numbers of incident edges the two vertices must also be such that after adding an edge between them:
  - Their total number of input internal edges is less or equal 2.
  - Their total number of output internal edges is less or equal 4.
  - Their total number of incident internal edges is less or equal 5.

- The generator then adds an internal edge from T_i to T_j. At the end of this first phase vertices T_0 and T_{n_t-1} are not yet connected and the rest of the DAG is not guaranteed to be fully connected. It could be composed of several disjoint sub-graphs.

- The second phase closes the DAG by adding one edge between vertex T_0 and each other vertex without a predecessor (except T_{n_t-1}), and one edge between each vertex without a successor (expect T_0) and vertex T_{n_t-1}. This second phase produces a connected DAG with one single source task (T_0) and one single sink task (T_{n_t-1}).

Of course, this algorithm works if and only if the specified number of internal edges n_e is less or equal the theoretical maximum: the n_t-2 non-artificial tasks have at most two input internal edges each, except task number 1 which has none and task number 2 which has at most one (coming from task number 1). The maximum number of internal edges is thus $0+1+2\times(n_t-4)=2\times n_t-7$.

We also excluded DAG with too few internal edges: applications with few or no inter-tasks dependencies are closer to a classical multidimensional bin packing problem than to our FPGA scheduling problem. We considered only graphs with at least one incident internal edge per non-artificial task, that is a minimum of n_t-3. This is the case, for instance, of a perfectly sequential application (or two independent sequential applications with one extra internal edge in one of the two sub-DAG).

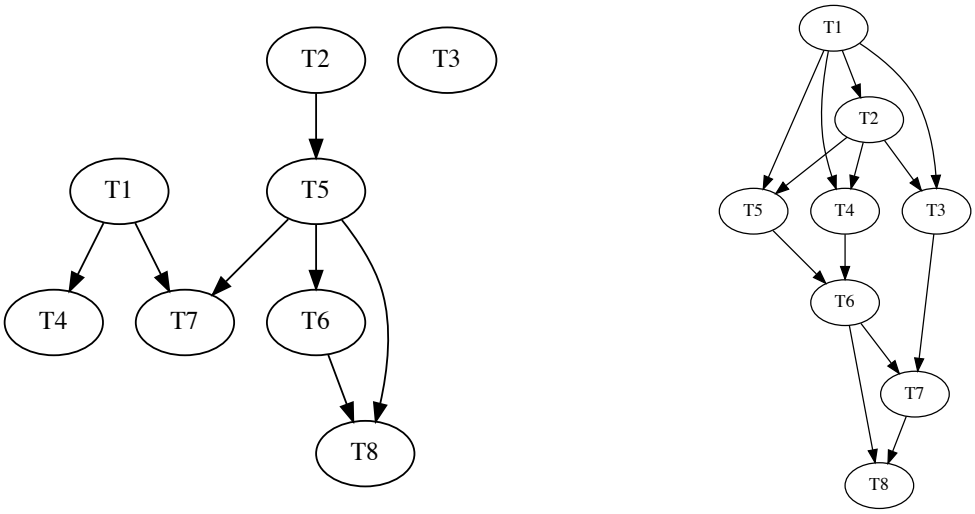
With this random generator we generated 37500 different instances with 6 to 30 tasks, 3 to 53 internal edges, and 5 to 61 total edges. In the following we will frequently split this benchmark in subsets according the number of tasks; we name "batches" these subsets.

Figure 5.1 shows two DAG, without the artificial source and sink tasks, of the $n_t=10$ tasks batch, one with $n_e=7$ internal edges (the minimum) and the other with $n_e=13$ (the maximum).

The goal of the constraints we integrated to our DAG random generator is to produce more realistic problem instances but they could also hide important aspects. In Section 5.4.4 we study the effect of relaxing these constraints.

5.4 Evaluation results

The target FPGA used for all evaluations was the Xilinx XC7S25 from the Spartan-7 family and we used a 3D model of it, with LE, RAM blocks and DSP resources. The XC7S25 FPGA is a relatively small FPGA but the complexity of the FPGA scheduling problem does not depend on the size of the target FPGA. Resources could as well be normalized and their quantities or consumptions represented as real numbers in the $[0,1]$ interval, it would not change anything. We could have used the Xilinx Virtex



(a) 7 internal edges

(b) 13 internal edges

Figure 5.1: Randomly generated 10 tasks DAG

Ultrascale P9 FPGA (the one used in F8 instances of Amazon Elastic Cloud), for instance, with similar results.

We compared 3 different implementations of one exact solver and two heuristics:

- A MILP solver written in C using the GNU Linear Programming Kit (GLPK) [7] and based on the MILP formulation presented in Section 5.2 and Appendix A.
- A *Slot* Java 8 implementation on a 64 bit Java Virtual Machine, version 1.8.0_201.
- A HEFT-NF Java 8 implementation on a 64 bit Java Virtual Machine, version 1.8.0_201.

HEFT-NF is the Next-Fit version of Heterogeneous Earliest Finish Time [104] heuristic already presented in Section 3.3.1.1. HEFT-NF is commonly used in the literature as a baseline for comparison [93, 100] because of its simplicity and high performance. To the best of our knowledge, a direct comparison with other heuristics is not possible without significantly denaturing them and biasing the comparison. Indeed, some of the works cited in Chapter 3 are based on simpler single resource models that are still valid for old FPGA without embedded DSP or on-chip RAM blocks. Other related works are based on design assumptions that conflict with the context of FPGA-based servers in cloud data centres. For instance, the contribution

in [110] is based on partial reconfiguration. In [42], independent tasks are packed, whereas we account for dependencies that partially constrain schedules.

All experiments have been run on a workstation with 2 sockets, 16 hyper-threaded cores per socket, that is, 64 logical CPUs, clocked at 3.5 GHz and with 64 GB of memory. Due to the very large set of instances and the run-times of the MILP solver, up to 60 runs have been launched in parallel. The memory monitoring shown that the memory was not a bottleneck.

For each problem instance the MILP solver produced an optimal solution and the two heuristics produced one approximate solution each. The schedule, the makespan and the tool's CPU User Time (CUT) have been recorded in a database. For the MILP approach a time-out of 24 hours has been set: for any instance exceeding this real time the solver has been stopped, and the instance has been excluded from the comparison with MILP. These "hard" instances have however been retained for comparisons not involving MILP.

5.4.1 The practical complexity of the FPGA scheduling problem

One of the first outcomes of our experiments is the practical complexity of the FPGA scheduling problem. 7486 out of 37500 problem instances were solved by MILP in less than the 24 hours time-out. All solved instances had between 6 and 15 tasks. We did not try to go above 15 tasks because the total CUT spent on the $n_t = 15$ tasks batch already represents about 6 days of cumulated CUT (not counting the 24 hours time-out of the aborted instances, plus the fact that the real time is always significantly larger than the CUT). Moreover, the proportion of instances for which the 24 hours time-out was exceeded increases with the number of tasks, which probably introduces a significant bias by keeping only "simple" instances. Continuing with larger instances would take huge computation times and produce more and more biased batches.

On the 7486 solved instances we estimated the practical complexity of the FPGA scheduling problem from the CUT taken by our MILP solver. The CUT were all measured with the `getrusage` function of the `glibc` library. As the complexity increases with the number of tasks, we analysed the distribution of the CUT independently for the different batches. Table 5.1 summarizes the observations for $6 \leq n_t \leq 15$, one row per batch, with the proportion of instances for which the MILP solver terminated before the 24 hours time-out (Solved), the total CUT spent on the solved instances (Total), the minimum, maximum, average, standard deviation and median of the CUT. All CUT are in microseconds. Zero values are due to the limited resolution of `getrusage`.

The CUT distributions of MILP solving are highly biased towards the minimum as can be seen for the $n_t = 15$ tasks batch (1187/1200 instances solved) and its histogram represented on Figure 5.2. There are more than 6 orders of magnitude

Tasks	Solved	Total	Min	Max	Mean	Std	Median
6	300/300	8.87E+05	0.00E+00	9.75E+03	2.96E+03	1.62E+03	3.05E+03
7	400/400	3.66E+06	0.00E+00	3.04E+04	9.15E+03	4.27E+03	8.53E+03
8	500/500	1.37E+07	2.63E+03	7.36E+04	2.75E+04	1.26E+04	2.56E+04
9	600/600	4.58E+07	1.07E+04	3.47E+05	7.63E+04	4.47E+04	6.60E+04
10	700/700	1.72E+08	2.70E+04	1.80E+06	2.46E+05	1.75E+05	2.02E+05
11	800/800	6.49E+08	7.90E+04	8.77E+06	8.11E+05	6.95E+05	6.62E+05
12	900/900	2.49E+09	2.09E+05	2.55E+07	2.77E+06	2.64E+06	2.02E+06
13	1000/1000	1.10E+10	5.41E+05	2.21E+08	1.10E+07	1.91E+07	6.38E+06
14	1099/1100	9.77E+10	1.11E+06	4.17E+09	8.89E+07	2.99E+08	2.53E+07
15	1187/1200	4.31E+11	3.44E+06	4.24E+09	3.63E+08	6.22E+08	1.28E+08

Table 5.1: MILP CPU user times in micro-seconds

between the non-zero minimum (0.0 milliseconds) and maximum (1.2 hours), and more than 95% of the instances fall in the first decile (0 to 7 minutes).

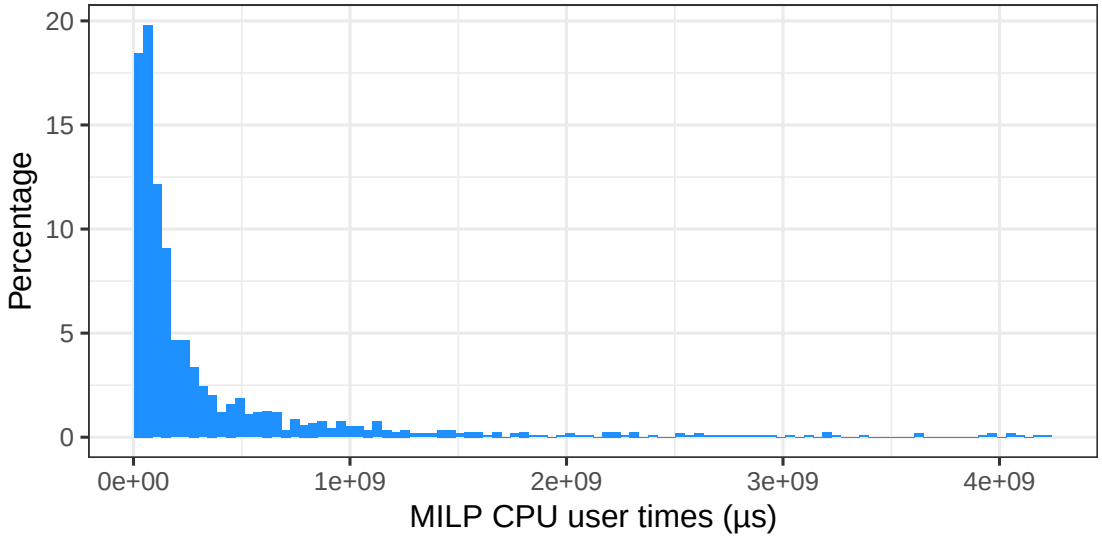


Figure 5.2: Histogram of MILP CPU user times for $n_t = 15$ tasks

Because of the extreme variability of the CUT and their bias in each batch towards the minimum, we decided to use the median instead of the less relevant average to compare the different batches. They are plotted in Figure 5.3 with a logarithmic scale.

Even if we can still not conclude about the real theoretical complexity of the FPGA scheduling problem, we clearly see that its practical complexity, measured on the CUT of a very well adapted exact solving technique, increases very rapidly with the

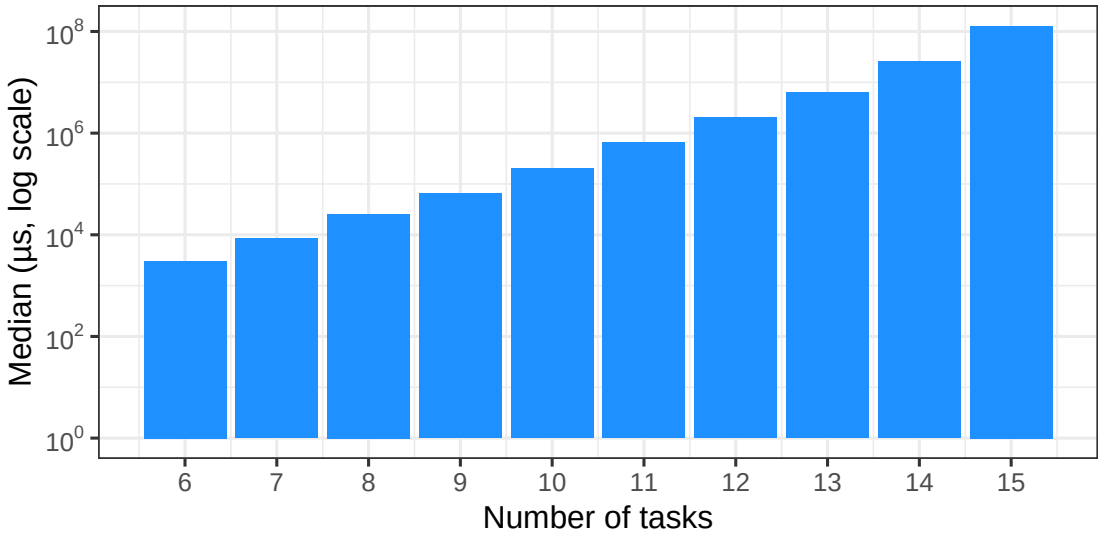


Figure 5.3: Median of MILP CPU user times vs. the number of tasks

size of the instances. This very rapid and apparently exponential complexity, at least for the numbers of tasks we could explore, again advocates for much faster, but still high quality heuristics.

5.4.2 Comparison of *Slot* and HEFT-NF quality

For the 7486 instances for which the MILP approach completed in less than the 24 hours time-out the criteria we retained to compare the quality of the schedules computed by *Slot* and HEFT-NF is the additional makespan they add to the optimum computed by MILP.

The Empirical Cumulative Distribution Function (ECDF) is commonly used to represent the distribution of quality metrics when comparing optimization techniques. We thus chose to plot the ECDF of the respective over-makespans, expressed as percentages of the MILP makespan. Let M_{ref}, M_s and M_h be the makespan of MILP, *Slot* and HEFT-NF, respectively, considered as random variables, we defined the over-makespans Δ_s and Δ_h as follows, where $x \in \{s, h\}$ for *Slot* and HEFT-NF:

$$\Delta_x = 100 \times \frac{M_x - M_{\text{ref}}}{M_{\text{ref}}}$$

And we plot:

$$\begin{aligned} \mathrm{ECDF}_x : \mathbb{R}^+ &\to [0,1] \\ \delta &\mapsto P\left(\Delta_x \leq \delta\right) \end{aligned}$$

The plots of the ECDF_s and ECDF_h for all batches are shown on Figure 5.4.

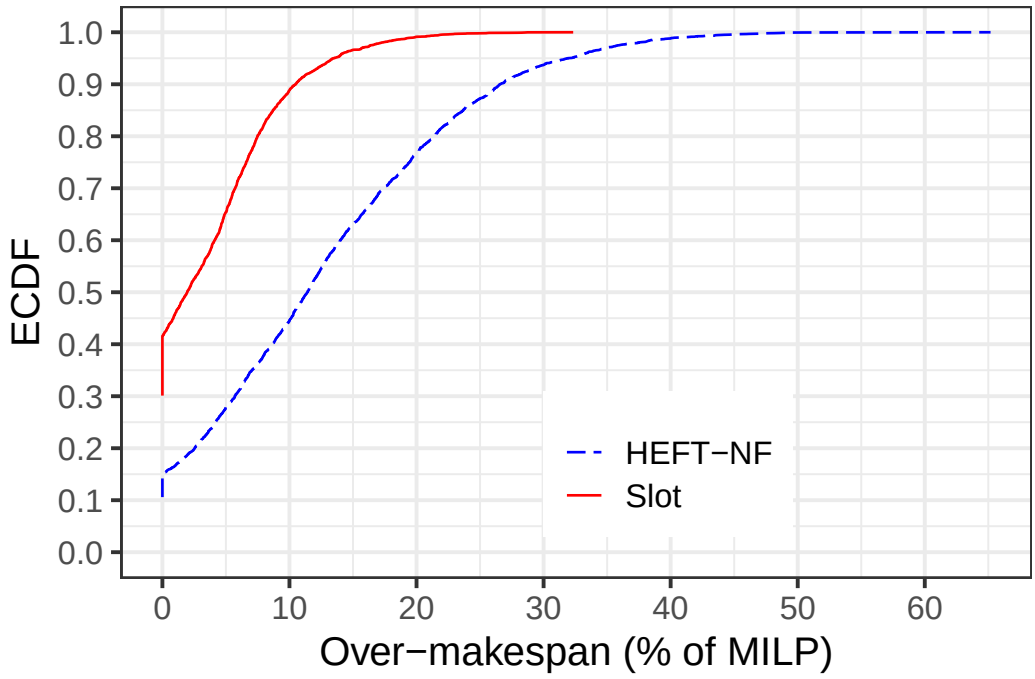


Figure 5.4: ECDF of *Slot* and HEFT-NF over-makespans vs. MILP, all batches

These ECDF curves clearly show that *Slot* outperforms HEFT-NF. Indeed, *Slot* over-makespan is less than 10% on more than 90% of the cases against about 45% of the cases for HEFT-NF. In order to analyse how this advantage over HEFT-NF depends on the number of tasks we also plotted separately on Figure 5.5 the individual ECDF curves per batch. The plots show that the *Slot* improvement over HEFT-NF is significant for all explored numbers of tasks with a dramatic advantage to *Slot* for small numbers of tasks. They also show that the quality of the two heuristics decrease when the number of tasks increases, which is not very surprising for a problem which practical complexity seems exponential.

To compare the performance of *Slot* and HEFT-NF on the instances for which the MILP solver exceeded the 24 hours time-out we cannot use an absolute optimum reference any more. We thus imagined a new heuristic, BEST, that simply consists in taking the best of the *Slot* and HEFT-NF solutions. The ECDF curves of the over-makespans of *Slot* and HEFT-NF vs. BEST for all batches are shown on Figure 5.6.

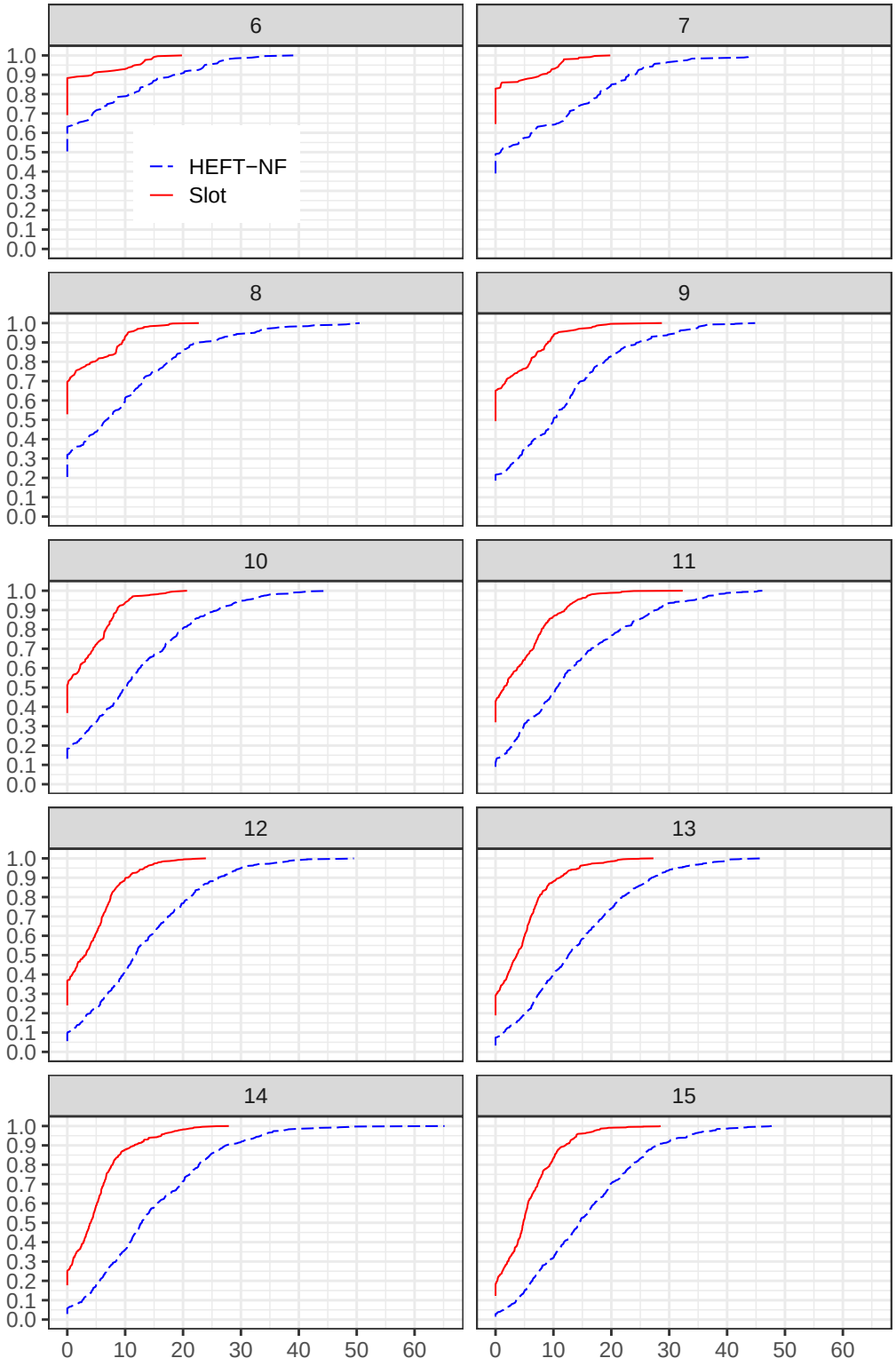


Figure 5.5: ECDF *Slot* and HEFT-NF over-makespans vs. MILP, separate batches

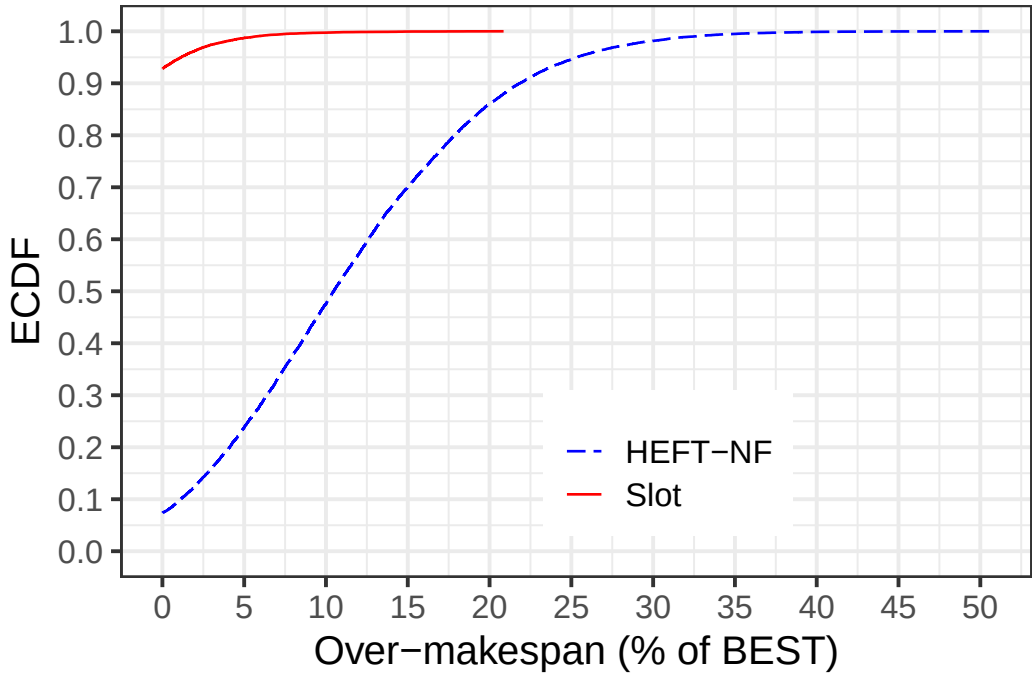


Figure 5.6: ECDF of *Slot* and HEFT-NF over-makespans vs. BEST, all batches

The individual ECDF curves per $16 \leq n_t \leq 30$ batch are shown on Figure 5.7. Again, these ECDF curves show that *Slot* outperforms HEFT-NF in most cases.

5.4.3 Comparison of *Slot* and HEFT-NF CPU user times

We did our best to implement MILP, *Slot* and HEFT-NF as efficiently as possible. MILP CPU user times have already been discussed in Section 5.4.1. Tables 5.2 and 5.3 summarize the observed CUT for *Slot* and HEFT-NF on a subset of the batches.

Tasks	Solved	Total	Min	Max	Mean	Std	Median
10	700/700	1.18E+08	1.00E+05	3.10E+05	1.68E+05	5.98E+04	1.30E+05
15	1200/1200	2.58E+08	1.30E+05	4.10E+05	2.15E+05	7.61E+04	1.70E+05
20	1700/1700	4.90E+08	1.60E+05	6.80E+05	2.88E+05	1.05E+05	2.30E+05
25	2200/2200	9.24E+08	2.00E+05	1.11E+06	4.20E+05	1.63E+05	3.50E+05
30	2700/2700	2.07E+09	2.90E+05	2.25E+06	7.68E+05	3.20E+05	6.90E+05

Table 5.2: *Slot* CPU user times in micro-seconds

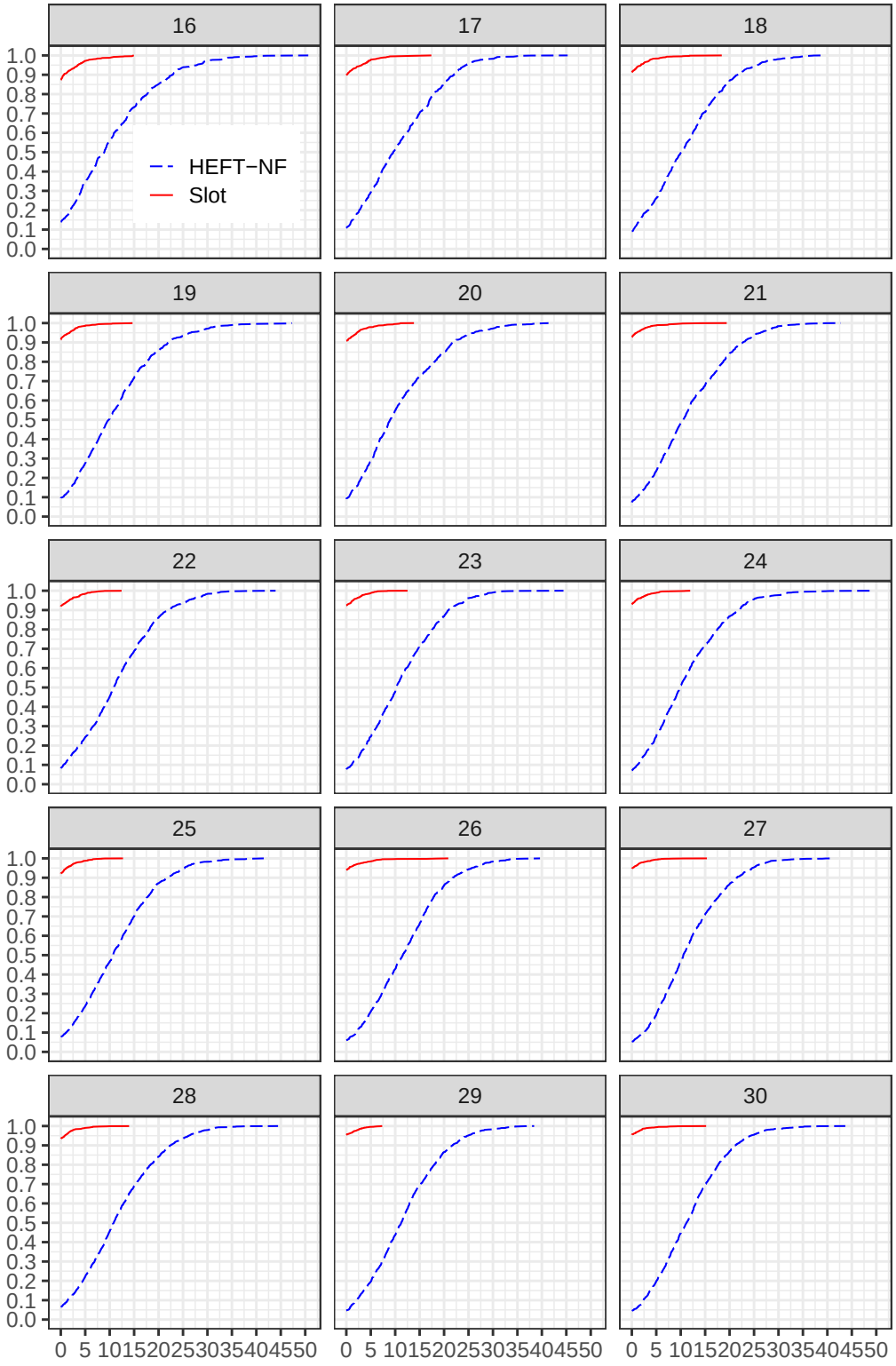


Figure 5.7: ECDF *Slot* and HEFT-NF over-makespans vs. BEST, separate batches

Tasks	Solved	Total	Min	Max	Mean	Std	Median
10	700/700	4.20E+07	4.00E+04	8.00E+04	6.01E+04	6.23E+03	6.00E+04
15	1200/1200	1.05E+08	4.00E+04	1.70E+05	8.72E+04	3.18E+04	7.00E+04
20	1700/1700	1.47E+08	4.00E+04	1.50E+05	8.63E+04	3.07E+04	7.00E+04
25	2200/2200	1.99E+08	5.00E+04	1.70E+05	9.05E+04	3.31E+04	7.00E+04
30	2700/2700	1.80E+08	4.00E+04	9.00E+04	6.66E+04	7.08E+03	7.00E+04

Table 5.3: HEFT-NF CPU user times in micro-seconds

Figure 5.8 shows the median of the CUT for the two heuristics as a function of the number of tasks.

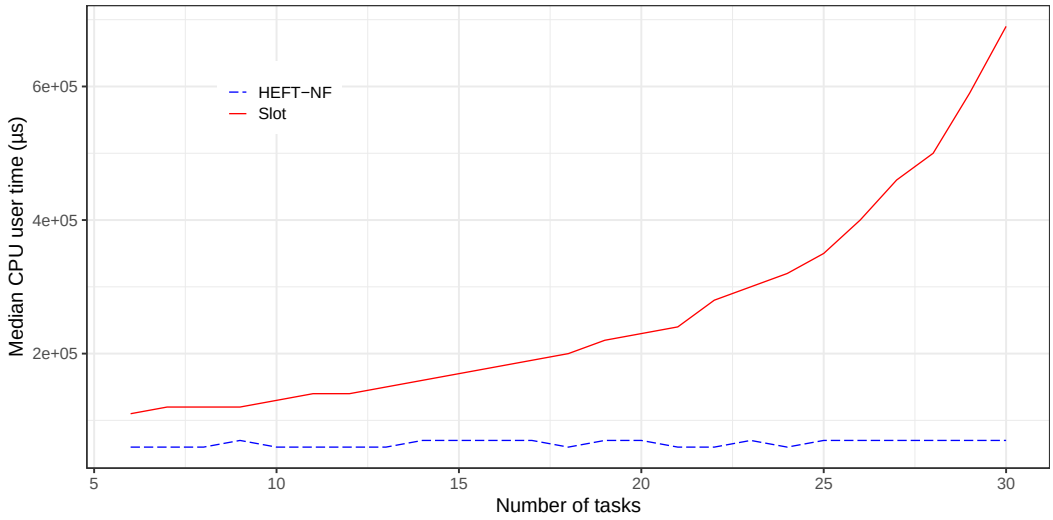


Figure 5.8: Median CUT of *Slot* and HEFT-NF vs. number of tasks

The minimum CUT of the two heuristics, observed for 6 tasks, are slightly larger than with MILP. This offset is due to the different programming languages (Java instead of C for MILP) and timing utility (`ThreadMXBean.getCurrentThreadUserTime` instead of `getrusage`).

Just like MILP, *Slot* has a growing complexity with the number of tasks. However, the median CUT for the 15 tasks instances is 170 milliseconds, that is, several orders of magnitude less than with MILP. Regardless the quality of our Java implementations, these CUT also show that HEFT-NF scales much better than *Slot* with the increase of the number of tasks. The HEFT-NF CUT is apparently almost constant, which means that a part of it is probably linear but with a too small contribution to the overall CUT to be visible on the instances we used. While the *Slot* CUT grows much faster with the number of tasks. For very large instances the *Slot* advantages

in terms of quality of the computed schedules will be counterbalanced by its larger run-times.

5.4.4 Problem instances with completely random DAG

As noted at the end of Section 5.3 the constraints we added to our instances generator may look arbitrary and could hide important aspects. In order to check this we relaxed the constraints on the DAG generation: the vertices can now have an arbitrary large number of incident edges. For each number of tasks $6 \leq n_t \leq 30$ and for each possible number of internal edges $0 \leq n_e \leq \frac{(n_t-3)\times(n_t-2)}{2}$ we generated totally random instances. As the number of $n_t \times n_e$ combinations is very large we generated 100 instances per combination where $6 \leq n_t \leq 15$ and only 10 per combination where $15 < n_t \leq 30$.

As with the first benchmark, in the $6 \leq n_t \leq 15$ batches we tried to obtain an optimum solution by MILP and we eliminated the 5940/36000 instances for which the 24 hours time-out was exceeded. For the larger instances we observed cases where the *Slot* real CPU time was significantly larger than with the first benchmark. We thus also added a 1 minute time-out to *Slot*, which eliminated 1273/62032 other instances. In total, this left a total of 60759 instances with completely random DAG, among which 30060 have an MILP optimal solution. The number of internal edges range from 0 to 378 and the total number of edges range from 5 to 380. Figure 5.9 shows two DAG (with source and sink tasks) of the 10 tasks batch, one with no internal edges and the other with the maximum: 28.

The plots of the ECDF_s and ECDF_h for all batches with a known MILP optimal solution are shown on Figure 5.10.

The individual ECDF curves per batch are plotted separately on Figure 5.11.

The plots show that the *Slot* improvement over HEFT-NF still exists for all explored numbers of tasks but it is less impressive than with the first benchmark.

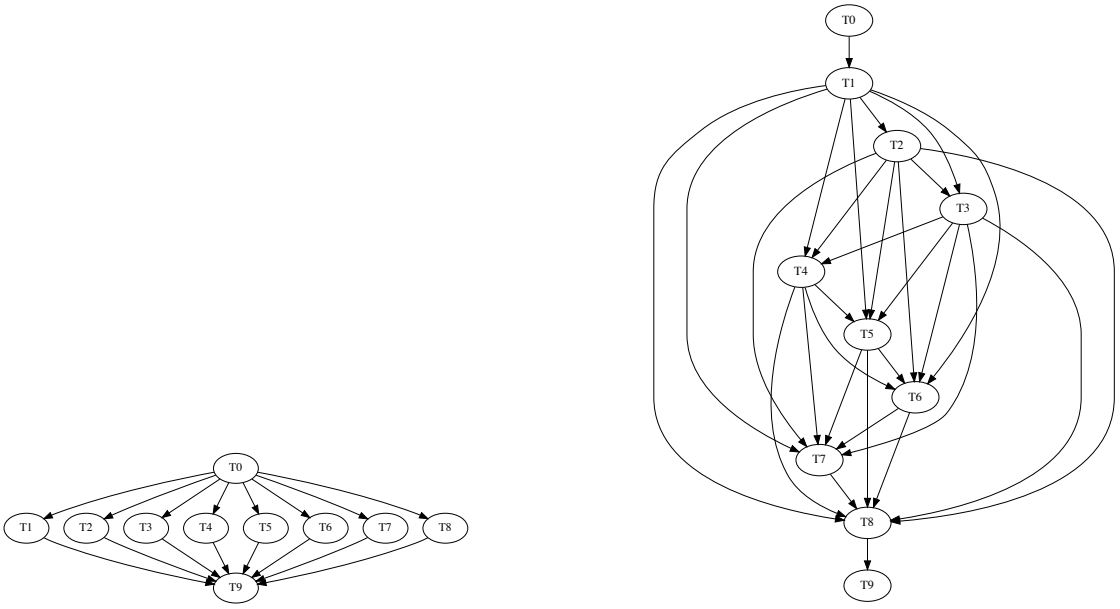
The performance of *Slot* and HEFT-NF on instances for which the MILP solver exceeded the 24 hours time-out are compared thanks to the same BEST reference heuristic we already used with the first benchmark. The ECDF curves of the overmakespans of *Slot* and HEFT-NF vs. BEST for all batches are shown on Figure 5.12.

The individual ECDF curves per $16 \leq n_t \leq 30$ batch are shown on Figure 5.13.

Again, the *Slot* advantage over HEFT-NF is visible but not as much as with the first benchmark.

5.5 Conclusion

In this chapter we first presented a Mixed Integer Linear Programming (MILP) formulation of our FPGA scheduling problem. The technical details of the implementation



(a) Zero internal edges

(b) 28 internal edges

Figure 5.9: Randomly generated 10 tasks DAG

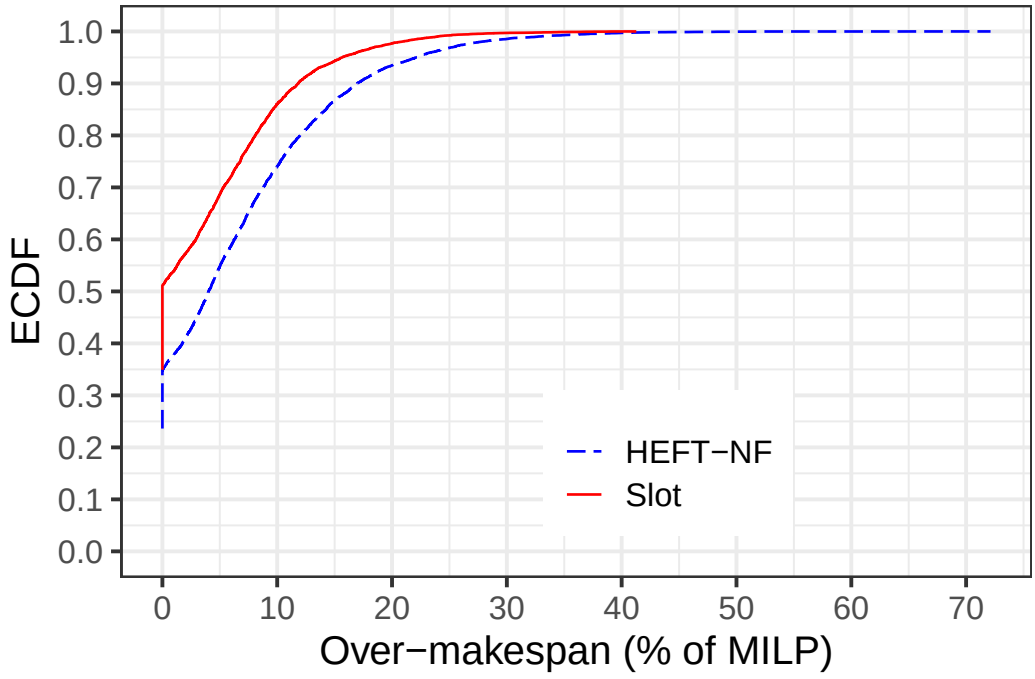


Figure 5.10: ECDF of *Slot* and HEFT-NF over-makespans vs. MILP, all batches

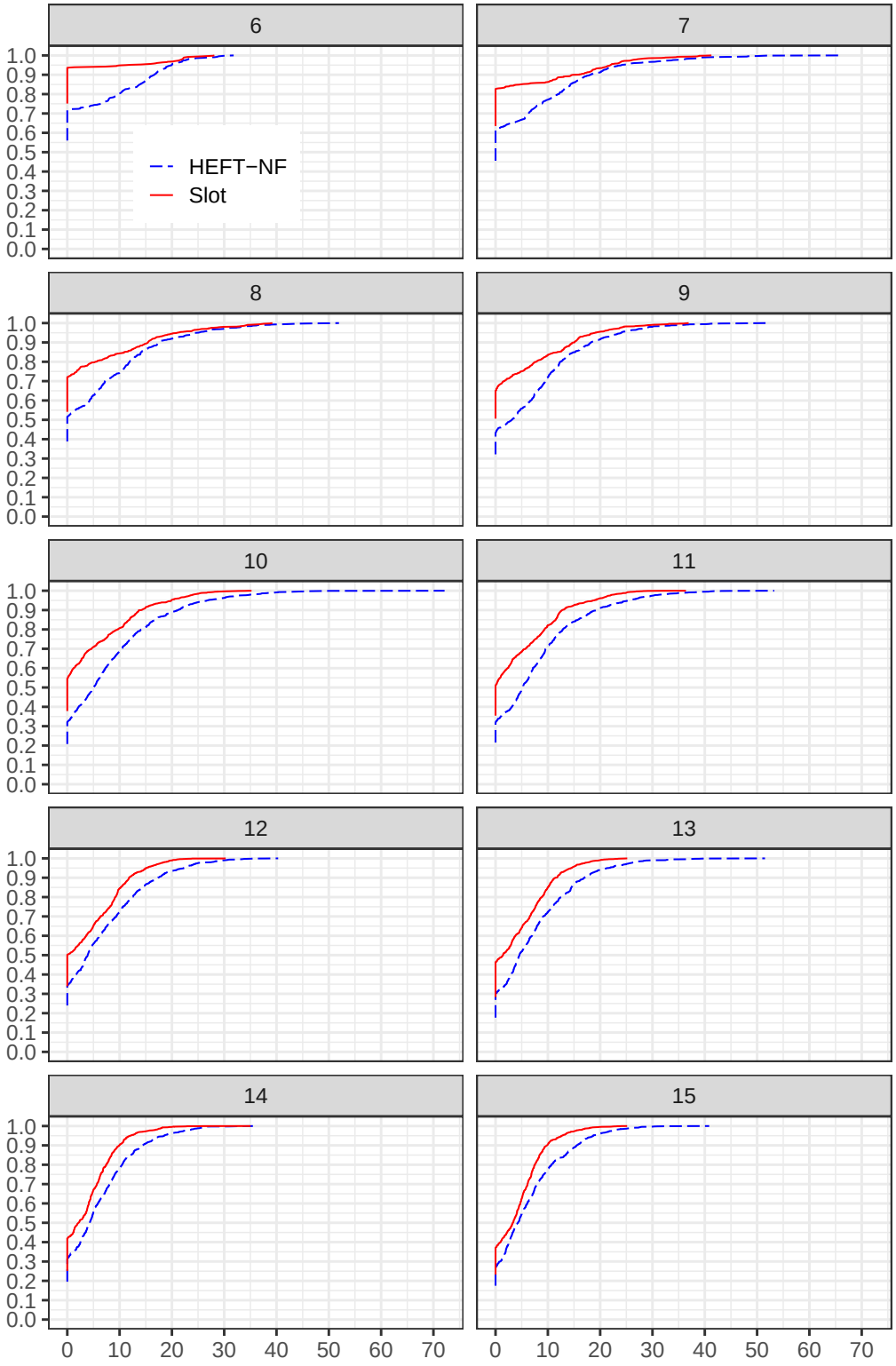


Figure 5.11: ECDF *Slot* and HEFT-NF over-makespans vs. MILP, separate batches

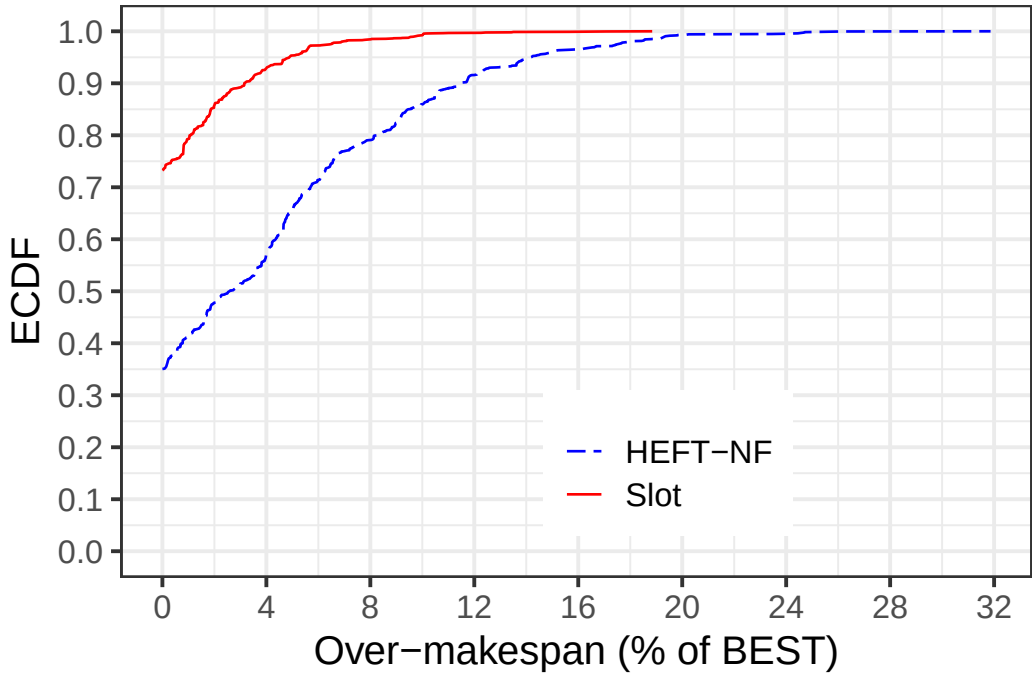


Figure 5.12: ECDF of *Slot* and HEFT-NF over-makespans vs. BEST, all batches

we designed based on this formulation are given in Appendix A. MILP is well adapted to this class of problems.

We then presented how we randomly generated a first large benchmark of thousands of problem instances with various numbers of tasks, edges and Directed Acyclic Graph (DAG) shapes. We used random generation because a statistical evaluation of our heuristics needed a large number of instances and such large numbers cannot be obtained from real-world use cases. For this first benchmark we constrained the DAG shapes to eliminate pathological graphs with too few or too many inter-tasks dependencies.

We then ran our MILP solver and our two heuristics (*Slot* and the reference existing work HEFT-NF) on this benchmark. The first outcome was the MILP run-times which are apparently exponential in the instances' size (number of tasks). This confirmed that we are facing a hard problem, if not theoretically, at least practically. Due to the very long run-times only the "smallest" instances could be solved by MILP. We used these optimal solutions to show how *Slot* outperforms HEFT-NF and is close to the optimal in a large proportion of the cases. Our metric for this comparison was the over-makespan added by the heuristics to the minimal makespan found by MILP. We however noted that the performance of the two heuristics decreases, still compared to MILP, when the size of the instances increases.

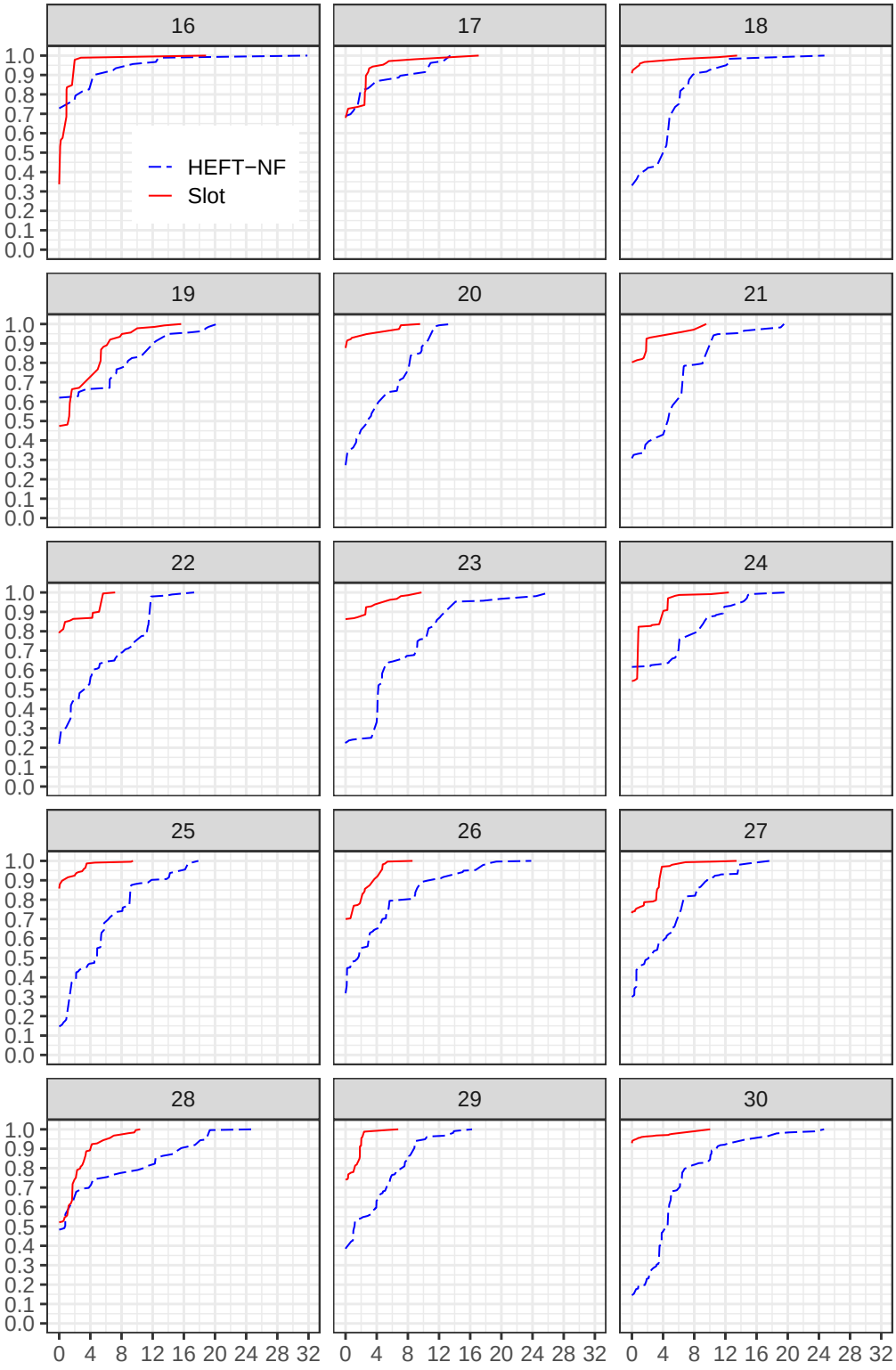


Figure 5.13: ECDF *Slot* and HEFT-NF over-makespans vs. BEST, separate batches

On larger instances for which we had no known optimal solution we compared our two heuristics to BEST, a kind of meta-heuristic which consists in taking the best of the two. There again, *Slot* shown a significant makespan advantage over HEFT-NF, even if, without an absolute reference, it was impossible to estimate how far from the optimum the proposed schedules were.

On the run-times point of view the advantage is clearly in favour of HEFT-NF, which CPU User Time (CUT) seams almost constant while the *Slot* CUT increases rapidly with the instances' number of tasks. Even if *Slot* CUT is several orders of magnitude less than that of MILP our experiments indicate that for very large instances *Slot*'s run-times will probably counterbalance its better makespans.

Finally we verified that the quality of the schedules computed by *Slot* are still better than the schedules computed by HEFT-NF on a second benchmark where all constraints on the shape of the DAG have been relaxed. We discovered that on theses instances *Slot* still offers a quality advantage over HEFT-NF but also that it sometimes exhibits much larger run-times than with the first benchmark, exceeding the 1 minute time-out that we set.

In conclusion we can say that *Slot* is definitely well adapted to small to medium size instances of the FPGA scheduling problem (e.g. 6 to 30 tasks), especially if the shape of the DAG is not too exotic. When the size of the instances increases its quality advantage over HEFT-NF is counterbalanced by its increasing run-times, again especially with unconstrained DAG shapes with very few or a lot of inter-task dependencies.

For a given host system where FPGA scheduling is needed, the choice of one heuristic or the other (or even the BEST mixed heuristic) depends on the operating conditions: size and shape of instances, rate and maximum latency of scheduling decisions.

Chapter 6

Integration to Model-Driven Engineering

We expect that *Slot* might also be interesting for the design of complex embedded system. Thus, in this Chapter, we apply *Slot* in another context than cloud data centers. For better supporting the design of embedded systems, we have integrated *Slot* into a Model-Driven Engineering (MDE) framework called TTool. TTool is a free and an open-source tool that supports several development stages with UML/SysML, e.g. requirements capture, analysis, hardware/software partitioning and (embedded) software design. Among various MDE tools, TTool was selected because it is lightweight, easily extensible, and TTool has already demonstrated to support signal processing applications [78]. The rest of this chapter is organized as follows. Section 6.1 presents Model-Driven Engineering and it introduces some related works. Section 6.2 describes the hardware/software partitioning stage of TTool (called "DIPLODOCUS"). Section 6.3 shows how we have integrated *Slot* into TTool/DIPLODOCUS and demonstrates this integration with a case study.

6.1 Model-Driven Engineering

Model-Driven Engineering (MDE) targets (high-level) embedded system modeling by offering dedicated models to capture heterogeneous hardware/software components. Models can represent an application, a platform and how an application can be mapped onto a platform. In addition, models can be transformed in order to generate executable models (for formal verification or simulation purpose) and executable code from high-level models. Thanks to their inner abstractions, models are expected to help focusing on the most important aspects of a system. With respect to FPGAs, we think that two important features shall be taken into account: hardware

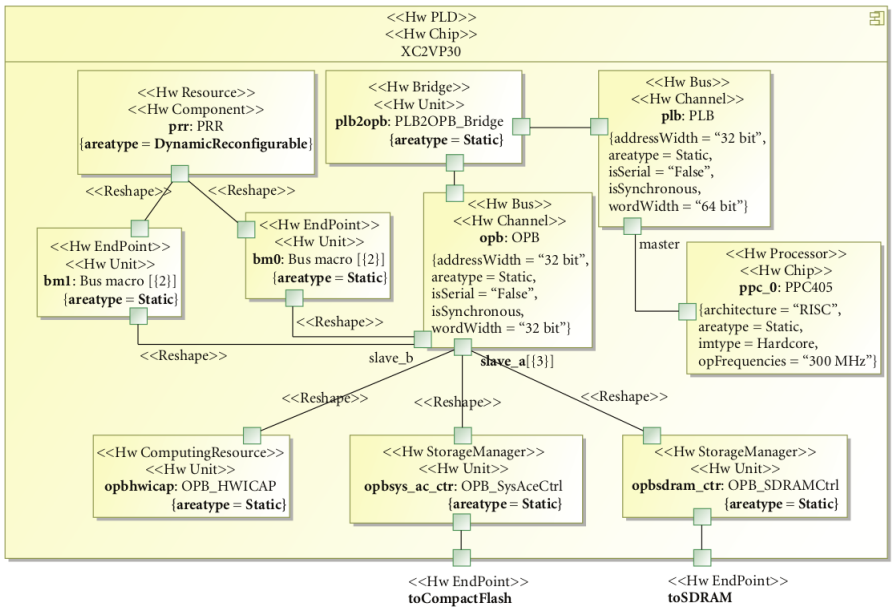


Figure 6.1: FPGA modelling in Gaspard and MARTE, from [95]

parallelism and dynamic reconfiguration.

Several modeling profiles and tools have been proposed to tackle to design of hardware and software components. MARTE (Modeling and Analysis of Real-Time and Embedded Systems) [8] is an OMG profile that extends UML for the design and the development of embedded systems. Such extensions particularly target hardware systems. Gaspard [103] is a MDE-based System on Chip co-design framework based on MARTE. One of their goals is the representation of parallelism, both for software and hardware aspects. In particular, Gaspard exploits the inherent parallelism of hardware elements or regular operators such as application loops. It is thus well adapted to represent FPGAs. Papers [95] and [94] suggest the use of MARTE models of GASPARD to generate an intermediate Register-Transfer-Level model that can be in turn transformed into VHDL code for synthesis. Figure 6.1 represents the modelling XC2VP30 Virtex-II Pro chip reconfigurable architecture. It is composed of several units, such as the Partial Reconfigurable Region (PRR), a storage manager, peripherals and a communication medium, the Bus Macro (BM). Reshape connections are used to connect the ports of the different block. Blocks defined in Figure 6.1 can contain sub-blocks giving more details about the master block. For instance, Figure 6.2 shows the internal composition in MARTE of the PRR block. The latter is composed of a hardware PLD connected through two ports to an Intellectual Property Interface (IPIF) module, namely a Xilinx wrapper for hardware buses. Finally, the FPGA model that can be built in MARTE and Gaspard can be very precise.

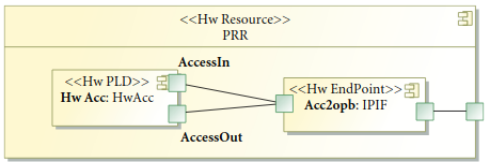


Figure 6.2: Internal composition of PRR block, from [95]

Such representations are interesting because they focus on the main FPGA features while remaining sufficiently precise to generate executable codes for FPGAs. Obvisouly, since the objective of *Slot* is not to generate FPGA code —*Slot* rather targets the **early design** of embedded systems—, low-level details of the platform can be left apart. The level of abstraction offered by TTool/DIPLODOCUS is definitely more adapted to *Slot* since TTool/DIPLODOCUS targets high-level decisions about the platforms and the mapping of functional aspects to the select platform. Having information about how tasks mapped to FPGAs could be efficiently scheduled is likely to support designers' decisions on mapping, or on architecture selection, thus being a building brick of Design Space Exploration engine (DSE) [74] [53].

6.2 TTool/DIPLODOCUS

6.2.1 TTool/Diplodocus overview

TTool/DIPLODOCUS *(DesIgn sPace exLoration based on fOrmal Description teChniques, Uml and SystemC)* [25] [13] targets the partitioning of Systems-on-Chip, i.e. finding the best candidate software and hardware architecture for executing a set of functions. TTool/DIPLODOCUS supports the Ψ-chart approach [48], namely an extension of the well known $Y-$chart approach [70]. TTool/DIPLODOCUS have several abstractions helping to keep models at a high-level of abstractions (e.g., value and type of data for the application, size and policy of cache memories for a CPU for the platform). Designs can be verified by a simulation engine [73] or a formal verification engine to check for e.g. liveness, reachability, scheduling and simulation code can be automatically generated.

According to the Ψ-chart approach, the partitioning of the system is done with four views, as shown in Figure 6.3:

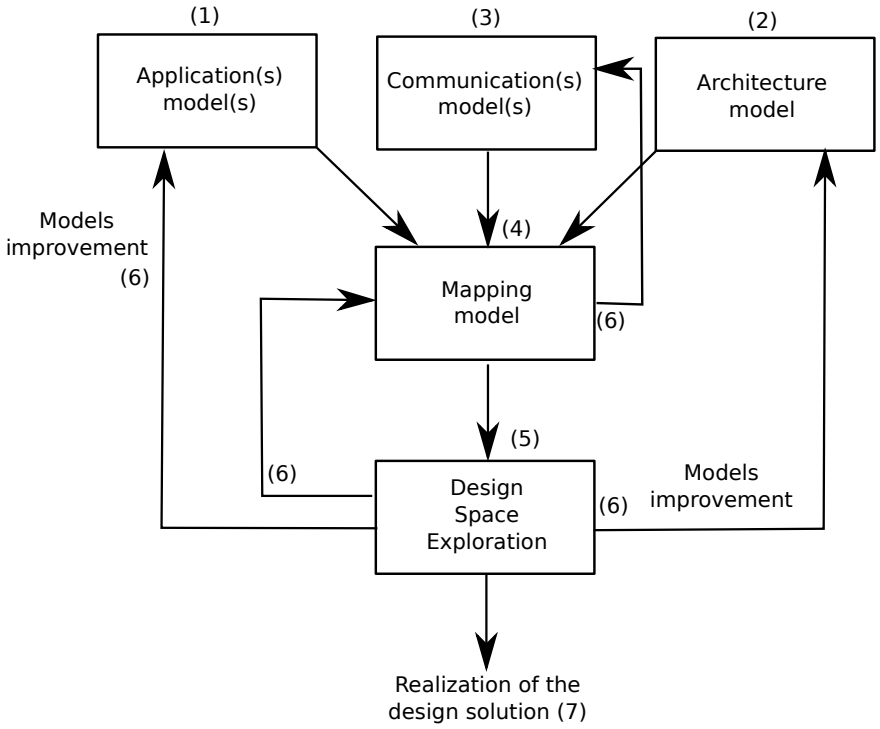


Figure 6.3: Ψ-Chart approach from [48]

1. *Application view:* functions of the system are modeled with SysML Blocks which can be interconnected by both data and control dependencies via ports. The internal behavior of each block is described by a UML Activity Diagram. In these diagrams, only the amount of data exchanged between blocks is modeled, whereas decisions that depend on the value of data can be expressed in terms of non-deterministic and static operators (i.e., non-deterministic choices). Algorithms are modeled with processing complexity operators intended to indicate the minimum and maximum number of operations (e.g. int, float, custom) required to complete an algorithm. These functional blocks are modeled independently from their respective implementation (hardware or software).

2. *Platform view:* the hardware elements that the system under design uses are modeled with different types of parametrized nodes. There are processing nodes like CPUs, DMAs and hardware accelerators, communication nodes like buses and bridges and storage nodes like memories. These interconnected resources are represented with UML Deployment Diagrams and they are characterized by performance parameters used for DSE purposes or for rapid prototyping. For example, a bus is defined by an arbitration policy and the size of data, among others, and a CPU by the number of cores, its clock

frequency,... These hardware node are highly abstracted with regards to the usual SystemC abstraction levels (TLM, CABA).

3. *Communication view:* in the context of the Ψ-chart approach, communications are modeled independently of the underline architecture or the application's functionalities. The communication models implemented in TTool/DIPLODOCUS are called Communication Patterns (CPs) [48] and they model the communication protocol at both physical and data-link layers (referring to the ISO/OSI network protocol stack). A CP describes the behaviour of a communication protocol with two different UML/SysML diagrams: Activity Diagrams to express the high-level algorithm of a protocol and Sequence Diagrams to represent precise exchange of signals.

4. *Mapping view:* The mapping model is built upon the platform model to which it adds the application functions. The latter are mapped to hardware nodes. A function mapped to a hardware accelerator becomes an ASIC while a function mapped to a processor becomes software. Communications between applications are allocated onto communication patterns that are themselves mapped to storage nodes and communication nodes. UML artifacts are used for mapping purpose.

6.3 Integration of *Slot* in TTool

Figure 6.4 shows how our heuristic could help designing HW/SW platforms of embedded systems by extending the Ψ-chart approach. At first, tasks and their dependencies are to be designed in the functional view. The architectural view should contain FPGAs processing elements for which availability of resources as well as dynamic reconfiguration time must be specified. Communications are modelled too. At mapping, a subset of the tasks may be mapped onto FPGAs. In the case of task-to-FPGA mapping, information such as HET, number of LEs, DSPs, EMBs, etc. shall be provided. After the mapping stage, our heuristic can be applied to each FPGA in order to deduce the best task scheduling, thus leading to a pre-scheduled mapping model.

We have integrated *Slot* to TTool/DIPLODOCUS through a plugin. TTool/DIPLODOCUS was already able to represent FPGAs and dynamic reconfigurations as well as performing simulations on tasks mapped onto the FPGA. Yet, tasks scheduling for FPGAs had to be hand-made. Given an application mapped onto a FPGA, *Slot* can now be applied to determine scheduling slots. This scheduling information can be transmitted to the simulation engine —for instance to verify that this selected scheduling

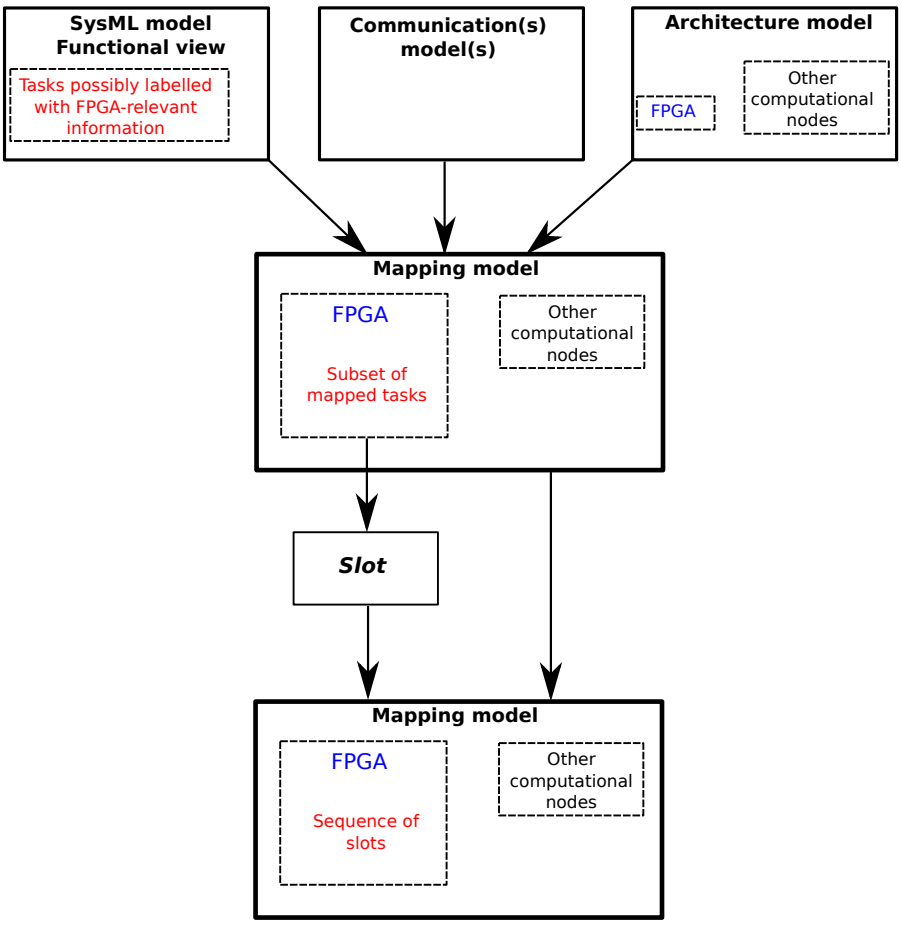


Figure 6.4: Integration of *Slot* in TTool/DIPLODOCUS and in Ψ-chart approach

works well with other parts of the system— or it can be used by the Design Space Exploration engine that can use information provided by slot to take further decisions.

We have modelled in TTool/DIPLODOCUS the application presented in Section 4.6. Figure 6.5 shows the application view. There, data dependencies among tasks are expressed through signals which are exchanged through *ports* (in light-blue). Each task can be labelled with the requests of resources. Figure 6.6 shows the mapping view. All the tasks of the application have been mapped onto the FPGA block. The latter is labelled with the information about scheduling-independent resources availabilities (e.g., LEs, DSPs and EMBs[1]) and the time required to perform the dynamic reconfiguration of the FPGA, as shown in Figure 6.7.

Figure 6.8 shows the internal behavior of task t_6. As we can see from signals, task t_6 receives data from tasks t_1, t_2 and t_5 and it sends data to tasks t_7 and t_8. The

[1]EMBs are called in TTool/DIPLODOCUS Block RAMs - BRAMs.

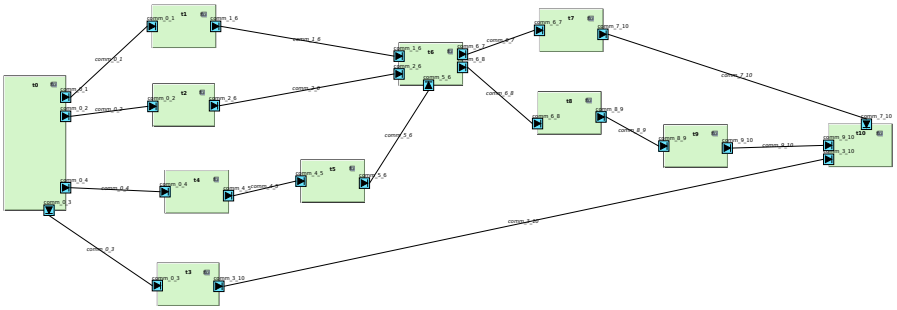


Figure 6.5: TTool - Application view

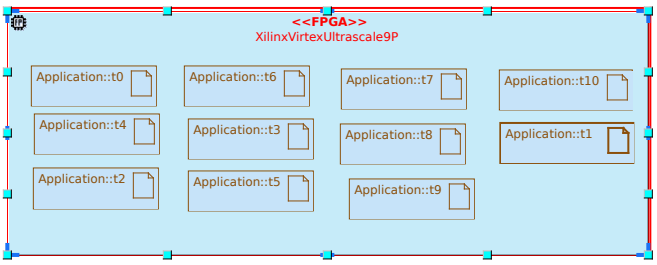


Figure 6.6: TTool - Mapping view

HET of task is modelled in terms of a number of integer operations (EXECI), which are 199 with regards to t_6. An EXECI corresponds to exactly one clock cycle of the FPGA, as it can be seen in Figure 6.7. Tasks are also modelled with their occupancy in terms of scheduling-independent resources.

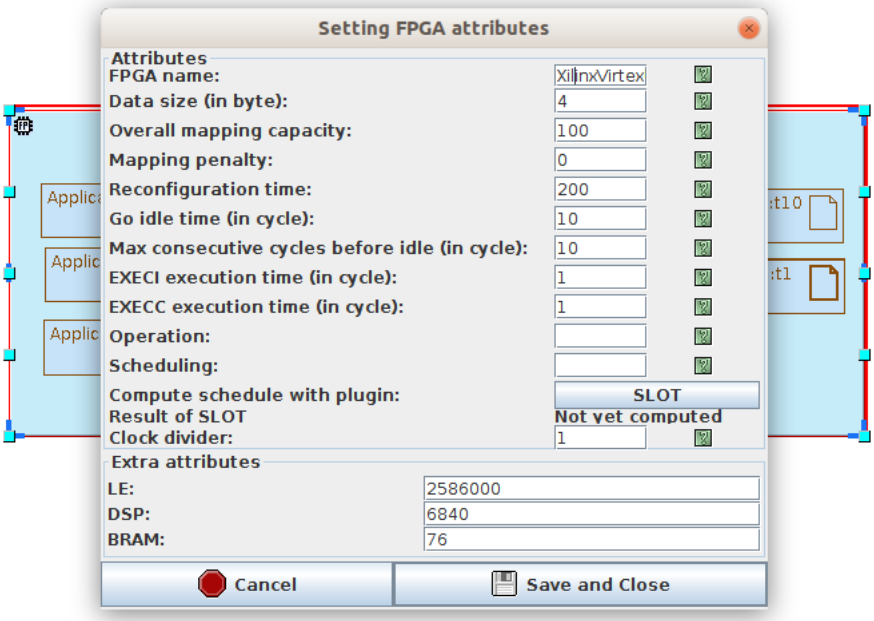


Figure 6.7: TTool - Parameters for FPGA block

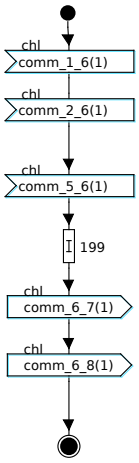


Figure 6.8: TTool - Activity Diagram for task t_6

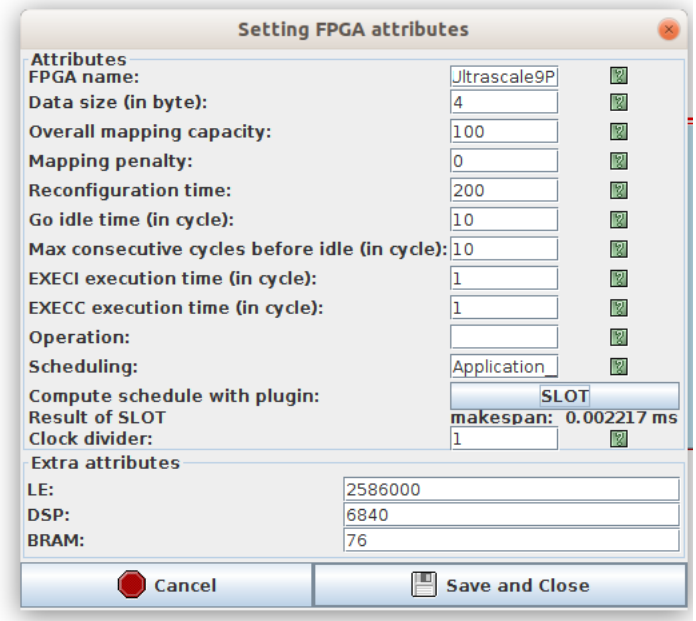


Figure 6.9: TTool - Application of *Slot* to the mapping view

Through the developed plugin, it is possible to use *Slot* to calculate slots and makespan, as shown in Figure 6.9. For this specific example, it returns the following slots: $[R,\ \{t_0, t_1, t_4\}]$, $[R,\ \{t_2, t_3, t_6, t_5\}]$, $[R,\ \{t_8, t_7, t_9, t_{10}\}]$, for a total makespan of 2217 ms. Thus, we have simulated the modelled application. For a better visualization, we invite the reader to consult the simulation trace, saved in HTML, which is present in [14].

6.4 Conclusion

Finally, we have tried to apply *Slot* in other context than cloud data centers. Thus, we have integrated *Slot* in a Model-Driven Engineering tool for the design of complex embedded system, named TTool/DIPLODOCUS. We have successfully modelled an application and we have thus obtained a simulation trace after the computation of slots through our approach. Such scheduling information can also be used by the Design Space Exploration engine of TTool/DIPLODOCUS in a wider context [74] [53]. Obviously, integrations with other domains may be meaningful, such as a middleware or an operating system for reconfigurable hardware (for instance: [24]).

Chapter 7

Future Work and Conclusion

7.1 Resume of the contribution

This manuscript has presented the scheduling of applications onto FPGAs. We have assumes that applications are composed of dependent tasks and that FPGAs are totally reconfigured. Our objective was the minimization of the application makespan.

In the first part of the manuscript, we have positioned our contribution in the context of cloud data centers. We have shown how recent cloud data center architectures integrate hardware accelerators, such as FPGAs, to better support the increasing requirements of applications in terms of processing power.

Then, the related work following the presentation of FPGAs and cloud data centers (i.e., Chapter 3) is divided in three categories. There are approaches based on exact mathematical formulations (e.g., MILP), which can compute the optimal solution, but they are slow to identify this solution. Second, there are approaches based on meta-heuristics (e.g., Genetic Algorithms, Simulated Annealing, Tabu Search, Ant Colony Optimization, etc.), which usually return a scheduling solution whose makespan is close to the optimum, but they are slow to return this solution (i.e., order of seconds/minutes for common applications). Third, heuristics addressing our problem are fast to execute (order of milliseconds) but as an average they do not return a result which is close to the optimum. Moreover, several related works capture applications and architectures using models which in our opinion are too abstract to be able to identify good (in terms of makespan) or valid scheduling solutions. Finally, the related work has shown that all these existing approaches cannot efficiently answer to our problematic.

Our approach can be seen as an intermediate between the presented approaches because it offers a good trade-off between the quality of the solution (in terms of makespan) and the run-time it requires to identify this solution. This approach is called *Slot*. *Slot* is based on an iterative process. It first considers all the graph and

the dominating task of this graph. From slots that can be built from this graph we use a scoring approach that tries to select the best slot. When a slot has been selected, we merge all the tasks of this slot together in the graph. Finally, this iteration outputs a sequence of slots which, after a final optimization, represents the proposed scheduling.

We have shown that Slot is efficient on a benchmark of 37500 pseudo-randomly generated graphs. We have also compared it to two other approaches: a MILP formulation[1] and an existing heuristic which has been adapted to better target the problem of FPGA scheduling (named HEFT-NF). HEFT-NF extends HEFT algorithm by maximizing the FPGA utilization. Our comparison evaluates both the quality of the results and the timing it takes computing the scheduling solution. We have shown that *Slot* outperforms HEFT-NF and is close to the optimal in a large proportion of the cases. Run-time of Slot is comparable to the one of HEFT-NF (i.e., tens of milliseconds), but HEFT-NF scales better with the number of tasks. This is different from MILP, which needs much more time to be executed. For example, the average run-time for 15-tasks graphs is in the order of tens of milliseconds for *Slot* and HEFT-NF, and hundreds of seconds for MILP.

In the last part of the manuscript, we have shown how *Slot* may be applied in other context than cloud data centers. We have integrated it in a model-driven engineering tool, named TTool/DIPLODOCUS, which supports the early design of embedded systems.

Figure 7.1 summarizes all the steps described in this conclusion with a schema.

7.2 Future work

Future work of this thesis targets different architectures than the one presented in Chapter 4. Indeed, we would like to apply the principles of *Slot* also in other types of situations that may be found in real projects. We remind the reader that the main steps of *Slot* are (i) privileging the dominating tasks, (ii) the generation of a set of candidate slots starting from a dominating task, (iii) evaluation of the most promising slot with respect to the overall makespan (for this we use a scoring-based system) and (iv) further optimizing the solution by compacting the selected slots. Thanks to the modular approach of *Slot*, we expect that by slightly adapting the different steps we can reuse *Slot* to other research problems.

Thus, we propose the six following directions:

1. Addressing scalability issues of *Slot*

[1] We remind the reader that MILP formulation always returns an optimal solution.

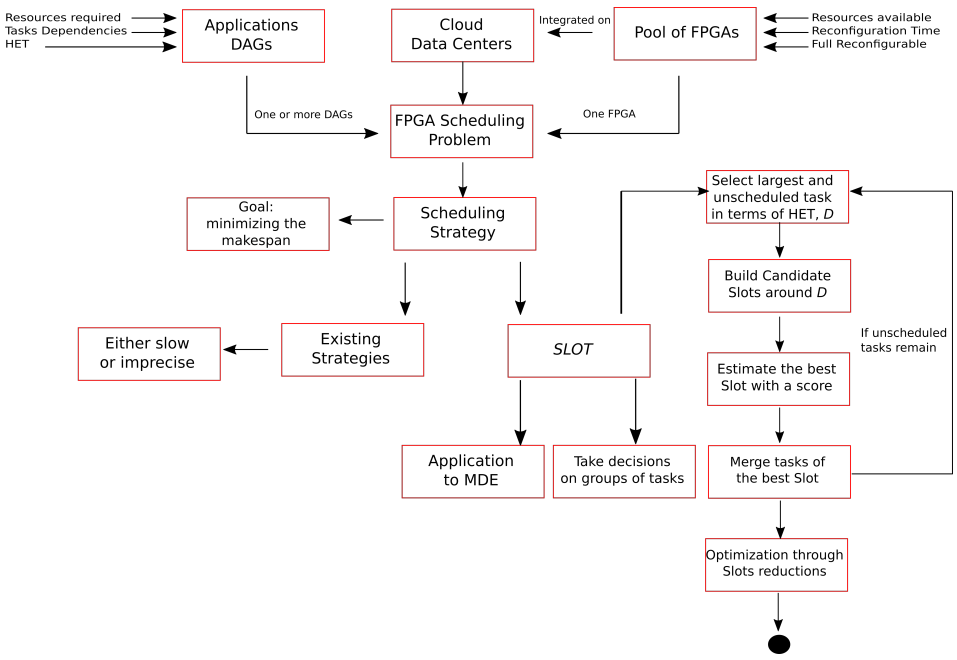


Figure 7.1: A schema which resumes the most important contents of this thesis

2. Scheduling of tasks that can request scheduling-dependent resources
3. Scheduling of both hardware and software tasks
4. Particular features of tasks
5. Exploiting the capabilities of remote FPGAs
6. Dynamic adaptation of scheduling in cloud

We have already started to tackle some of these directions.

7.2.1 Addressing scalability issues of *Slot*

From the complexity discussion in Section 4.6.5 and the evaluation in Chapter 5, we have noticed that *Slot* does not scale very well with the number of candidate slots, even if data-dependencies and resources availability limit the number of candidate slots. This complexity issue is particularly true for the following graph structure. For instance, the graph shown in Figure 7.2 is composed of three parallel *chains* (namely c_0, c_1 and c_2) of n tasks, with a total of 300 tasks (plus artificial source and sink). Let us imagine that the dominant task is the first task of the second chain, namely $c_1 t_0$. We have already seen a similar situation in the research problem explained in Section 1.4. There, we have understood that, for instance, a slot composed of

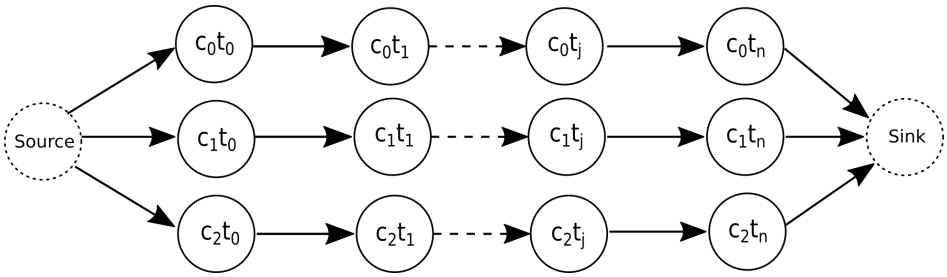


Figure 7.2: DAG composed of three parallel *chains* (namely c_0, c_1 and c_2) of n tasks

the dominating task c_1t_0 and a *far* parallel task, such as c_0t_n, is convenient only in particular conditions. More specifically, this slot should bring a great advantage from total makespan point of view, because all the potential parallelism between other tasks of $chain0$ and other tasks of $chain1$ would be removed (as in Figure 1.2). This means, for example, that the HET of c_1t_0 and c_0t_n should be similar and it should also be higher than the HET of the other tasks. So, a consequence of this could be that taking into consideration slots which are composed of tasks which are far from each other, could probably be avoided in most situations. As a conclusion, a part from few border cases[2], the scoring-based system can already handle such situations, but at the cost of an extra computation time. Thus, the main benefit from slot limitation is from run-time point of view but there are also cases for which the scoring system will not find the good solution when two concerned tasks are far from each other. But, again we expect that it mostly impacts the run-time and not the quality of the solution.

Thus, given a dominating task, whose distance from source, for instance calculated through Breadth-First Search, is d_x, we could consider only candidate slots composed of tasks whose distance from source is $d_x \pm k$. Thus, the new research problem to tackle would be to efficiently pre-analyse the DAG to calculate k. Very likely, the calculus of this parameter shall consider the average parallelism of the DAG, $HETs$ of tasks and the standard deviation among them.

7.2.2 Scheduling of tasks that can request scheduling-dependent resources

7.2.2.1 Recall about scheduling-dependent resources

As we have already explained in Section 4.4, we have classified resources in two types: scheduling-independent and scheduling-dependent resources. This section contains a reminder of the differences between them. A **scheduling-independent**

[2] An example of border case, with respect to Figure 7.2, is when the HET of $chain2$ is higher than the sum of the $HETs$ of $chain0$ and $chain1$.

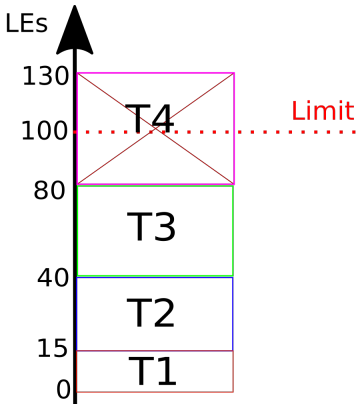


Figure 7.3: Occupancy of the FPGA with regards to the number of used Logic Elements (LEs). Tasks $T4$ cannot be part of the same slot of $T1$, $T2$ and $T3$.

resource is a resource which is exclusively assigned to a task for the entire lifetime of the slot that contains the task. Thus, requests for a given scheduling-independent resource (such as LEs) within a slot can be handled simply by adding them together. Yet, if the total amount of requested resources exceeds the availability (for a given FPGA) or all the tasks requesting the resources cannot be part of the slot. This is shown in Figure 7.3, in which task $T4$ cannot be part of the slot that already contains $T1$, $T2$ and $T3$.

A **scheduling-dependent resource** is a resource which is assigned to a task for the entire task's lifetime. The latter is always less or equal the entire lifetime of the slot that contains the task. Tasks whose requests of a scheduling-dependent resource exceed the physical limit for a given FPGA, such as $T1$, $T2$ and $T3$ in Figure 7.4, cannot be executed **in parallel**. However, differently from scheduling-independent resources, they can be part of the **same slot** by introducing a delay, as shown in Figure 7.5.

7.2.2.2 Architectures

Figure 7.6 shows a possible architecture that includes scheduling-dependent resources.

Thus, in addition to the architecture presented in Chapter 4, the platform in Figure 7.6 includes an external DRAM that is connected to the FPGA through a shared bus.

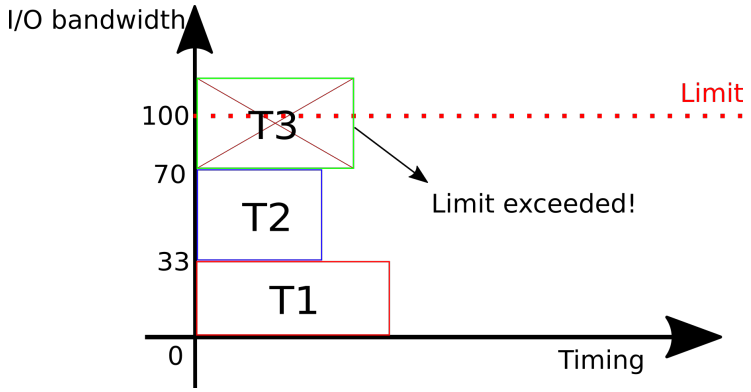


Figure 7.4: Occupancy of the FPGA with regards to the required I/O bandwidth. Tasks $T3$ cannot run in parallel with tasks $T1$ and $T2$.

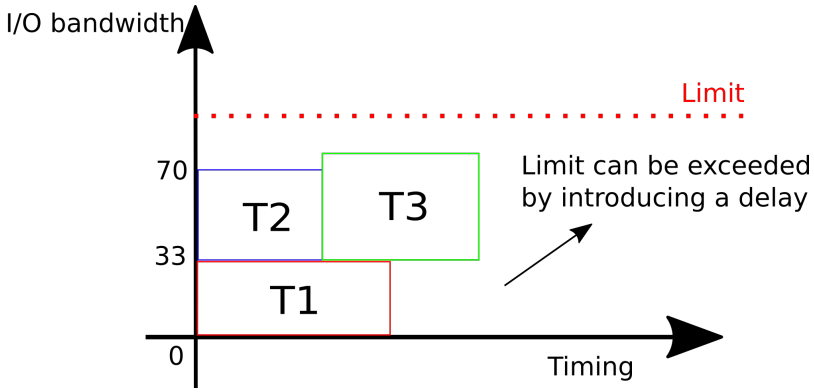


Figure 7.5: Occupancy of the FPGA with regards to the required I/O bandwidth. Tasks $T3$ cannot run in parallel with tasks $T1$ and $T2$, but they can be part of the same slot by introducing a delay

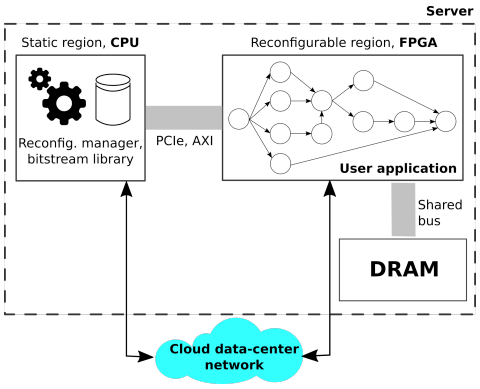


Figure 7.6: A possible platform that includes scheduling-dependent resources. In this respect, we can notice the presence of a DRAM, external to the reconfigurable chip, and a shared bus to this DRAM

Task	LEs	Badwidth$_{DRAM}$	HET	WCET$_{writing}$
t_1	50%	100%	100 [u]	50 [u]
t_2	50%	100%	80 [u]	50 [u]
t_3	50%	100%	20 [u]	80 [u]

Table 7.1: Resources requests of tasks of Figure 7.7

Tasks read the input data from this external memory and also write the results of their computations in this external memory. We remind the reader that, in Section 4.2, we assumed that the time to read, write and transfer the input/output data for a task in different memory locations was included in the HET of the task. Moreover, we have also assumed that all communications were performed without memory contention.

To better consider these architectures, we need to define how inter-task communications work. Let us assume that two tasks, t_1 and t_2, exchange data but are part of different slots. Since t_1 and t_2 are executed in different slots, they have to exchange data through the external DRAM. If t_1 and t_2 were to be executed in the same slot, they could use instead the Embedded Memory Blocks of the FPGA chip. The current version of *Slot* could consider the latter case (i.e., forcing two tasks to exchange data through the EMBs of the FPGA) by merging t_1 and t_2 as one unique task $t_{1,2}$ (so, t_1 and t_2 are mandatorily placed in the same slot). Based on this, we could imagine defining a new version of *Slot* that could decide between using EMBs or external memories to exchange data between tasks. Meanwhile, the workaround is running *Slot* on two different application models (t_1 and t_2 merged or not) and considering only the best one. Of course, this could increase the complexity of *Slot*.

To use scheduling-dependent resources we would also need to define other parameters. For instance, the shared bus and the DRAM could be simply captured with one unique resource: the maximum bandwidth with the external DRAM. Consequently, tasks should be labelled with the requested read and write bandwidth and the Worst-Case Execution Time (WCET) of this communication.

7.2.2.3 Motivating example

As we have seen in Section 7.2.2.1, when the sum of scheduling-dependent resources exceeds those offered by the architecture, tasks can still be part of the same slot but delays have to be introduced. This delay introduction raises a new issue: *with respect to the overall makespan, is it better to introduce a delay or to split the slot?* Let us consider the example in Figure 7.7. Resources needs of task is given in Table 7.1. For simplicity, let us assume that tasks write the results of their computations on an external DRAM, whereas the reading is instantaneous. The research problem can be summarized in the choice of the best scheduling between:

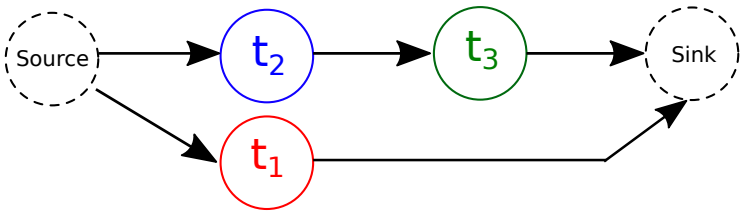


Figure 7.7: 5-tasks DAG

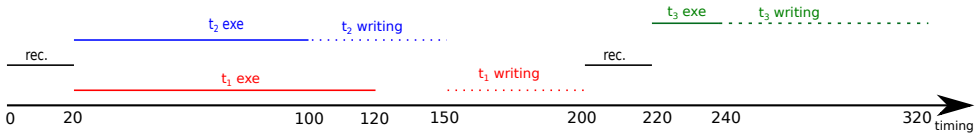


Figure 7.8: Gantt diagram for **Scheduling 1**

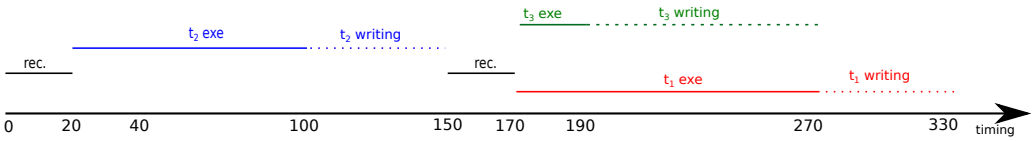


Figure 7.9: Gantt diagram for **Scheduling 2**

- **Scheduling 1:** $[R,\ \{t_1, t_2\}]$, $[R, \{t_3\}]$
- **Scheduling 2:** $[R,\ \{t_1, t_3\}]$, $[R, \{t_2\}]$

Let us view the two different scheduling with Gantt diagrams. Total reconfiguration time T_R is fixed to 20 units of time. Since t_1 and t_2 have the highest HETs, **Scheduling 1** better exploits the parallelism between the **execution** of tasks. However, as shown in the Gantt diagram of Figure 7.8, this involves a contention while writing the result of the tasks in the DRAM. Thus, the dominating task t_1 must wait 30 units of time before being able to access the DRAM. In **Scheduling 2** the situation is different, because in the first slot task t_3 has already finished to write its results to the memory when the dominating task t_1 can finally write its results to the DRAM, as shown in Figure 7.9. Finally, the makespan of **Scheduling 2** is slightly higher than the makespan of **Scheduling 1**. However, *Slot* is currently not able to correctly evaluate even simple situations such as the one in our example. Indeed, the scoring-based system introduced in Chapter 4, would have definitely chosen **Scheduling 1**, because the first computed slot contains the two tasks with higher HET, namely tasks t_1 and t_2. Actually, we have seen that both **Scheduling 1** and **Scheduling 2**

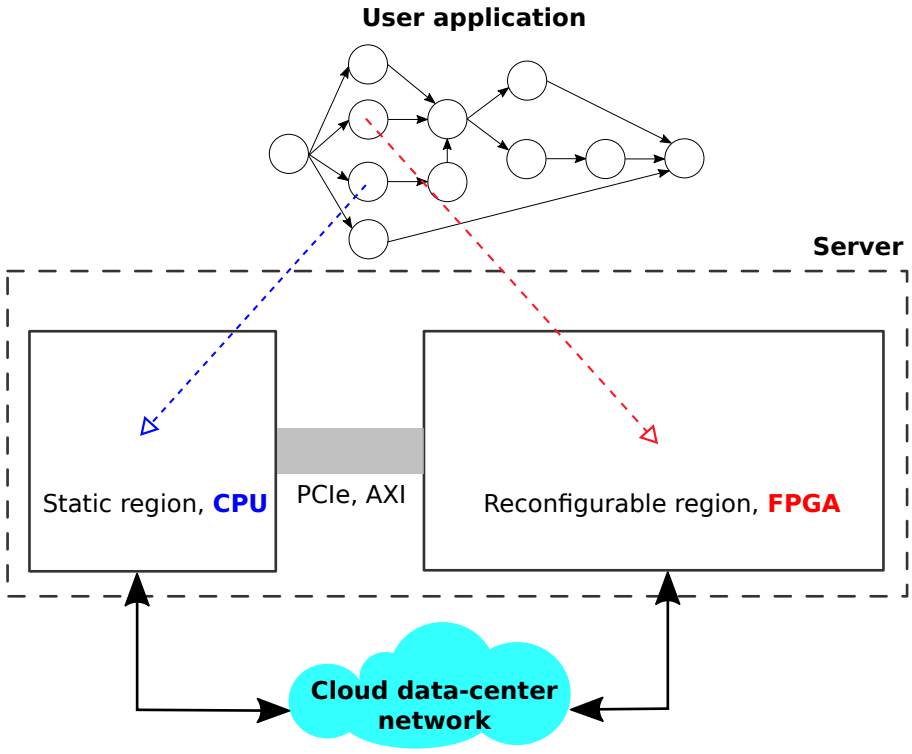


Figure 7.10: The architecture of a modern FPGA-based server - scheduling of both hardware and software tasks

have similar makespans. So, the scoring-based system did not recognize this similarity.

So, currently *Slot* lacks the concept of contention to shared resources. For this reason, it considers scheduling-dependent resources as they were scheduling-independent and this could lead to erroneous slots estimations.

7.2.3 Scheduling of both hardware and software tasks

We would like to apply the principles of *Slot* also in the context of heterogeneous scheduling, namely the scheduling of applications whose tasks can either have hardware implementation and therefore be executed on an FPGA, or a software implementation and so being executed on a processor (CPU). We can thus reuse the platform presented in Chapter 4, but now the CPU can be used to run some of the tasks that compose the application graph, as shown in Figure 7.10. A processor can execute only a maximum number of tasks which is limited by its number of cores. Each software task requires exactly one core of the CPU. The latter can be seen as

a scheduling-independent resource. Similarly to hardware tasks, also software tasks must be labelled with an execution time, named Software Execution Time (SET). We assume that this SET also includes the communication to the memory.

In Chapter 4 we have forbidden the pre-emption of hardware tasks during their execution on an FPGA. The pre-emption of a hardware task would have implied a total FPGA reconfiguration. We would like to allow the pre-emption of software tasks because of the *low overhead* of context-switching[3] with regards to the timing taken to reconfigure an FPGA. For instance, CPU overhead is in the order of microseconds for recent processors running a recent Linux kernel [107].

7.2.3.1 Challenges

We have identified three peculiarities of software tasks that *Slot* cannot take into account efficiently:

- Software tasks can run while FPGAs are being reconfigured;
- Software tasks can be pre-empted.

Slot as presented in Chapter 4 Cannot estimate how much software and hardware tasks can run and where this parallelism is located (in terms of dependencies). In other words, *Slot* is unable to exploit the flexibility given by software tasks: it could only consider them as they were hardware tasks, so in a non-efficient way.

Let us consider the graph in Figure 7.11. For convenience, tasks are graphically labelled with the information about their nature (i.e., software or hardware) and their execution time (either HET or SET). For the sake of simplicity, we do not consider resource requests for hardware tasks for this example. In this graph, task t_3 has the highest execution time, but it cannot be the dominating task because it is a software task. Since t_3 is a software task, its execution time could be *masked* by other hardware tasks executed in parallel to it. For instance, in this graph, t_1, t_2, t_4, t_5 and t_6 could be executed in parallel to t_3. In addition, t_3 can also run during the dynamic reconfigurations of the FPGA, if any. Moreover, the other software task t_8 is only parallel to one hardware task: t_7. So only 50 units of time of t_8 can be *masked*.

This leads to the introduction of a concept which is specific to software tasks only, namely the concept of **True Software Execution Time** ($TSET$). The $TSET$ of a task t, noted as t_{TSET}, is the time of a software task that cannot be parallelized with the execution of hardware tasks. In general, the more the ratio $TSET/SET$ tends to 0, the more the execution of a software task can be masked by the execution of other hardware tasks. Thus, we could think that the scheduling algorithm could ignore *ignore* t_3 or, at least, it could give it a less important weight, because its execution

[3] We remind the reader that the context-switching is the process of storing and restoring the state (context) of a process so that execution can be resumed from the same point at a later time.

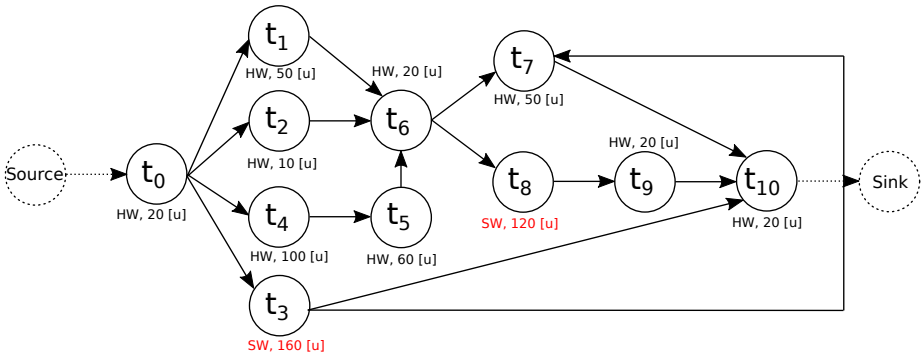


Figure 7.11: 13-tasks DAG. Each task has been labelled with the information about the execution node that will execute it (either the CPU or the FPGA)

can be in any case be put in parallel to the execution of other hardware tasks. So, for software tasks, the idea of dominating task is probably not as important as for hardware tasks.

Yet, in this example, the calculation of t_{3TSET} and t_{8TSET} has been trivial because the set of hardware tasks which are in parallel to t_3 is totally disjoined by the set containing those parallel to t_8. However, this situation is not so frequent in concrete applications and so the calculus of the $TSET$ of a task may not be easy to define: this is still an open issue. Obviously, the current version of *Slot* is neither able to evaluate how much the software tasks are parallel to the hardware tasks, nor it is able to take into account the flexibility given by the pre-emption or the concept of $TSET$.

7.2.4 Particular features of tasks

Another possible future work is to consider advanced features of tasks. We therefore discuss about two possible directions.

7.2.4.1 Addressing duplicated tasks

An interesting discussion is that *Slot* works better when all the tasks that compose the application graphs are different from each other. Indeed, in case the application graph contains multiple instances of the same task at different places of the graph, we think a better way of computing slots should be defined. Thus, it may be better to put the same instances in the same slot. But currently, *Slot* cannot consider duplicated tasks, so it considers multiple instances of the same tasks as if they were different tasks. So currently it lacks information to better handle this situation.

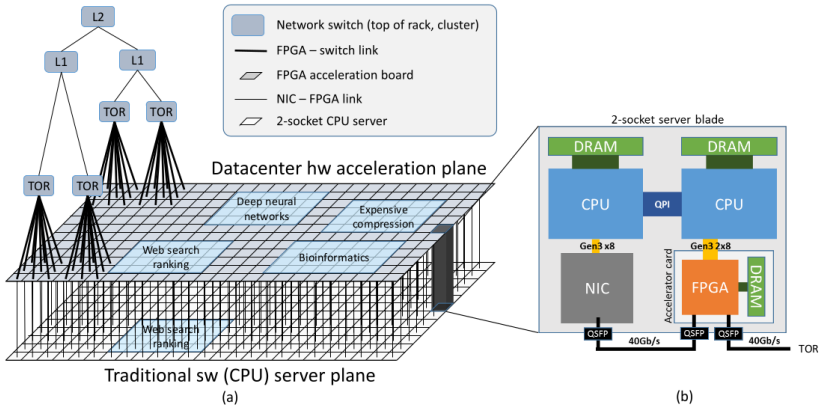


Fig. 1. (a) Decoupled Programmable Hardware Plane, (b) Server + FPGA schematic.

Figure 7.12: Microsoft Catapul cloud data center architecture, from [33]

7.2.4.2 Multiple implementations

Some related works, such as [42] and [27], consider multiple implementations for the same task, and they let the scheduling algorithm deciding which one would bring more benefits to the objective function. As explained in [71], a hardware task can have multiple implementations. For instance, let us consider a task which executes a function with a sequential algorithm. Let us now assume that this implementation can be parallelized. In this second case, task will execute in less time, but very likely it will require more hardware resources. This future work would require to integrate in *Slot* the concept of Design Space Exploration. Currently, what we could do is running n times *Slot* for n different application models (each one with a different implementation of a task) and comparing the results. But unfortunately, if several tasks of the application present multiple implementations, this leads to evaluate many all the combinations: this approach does not scale.

7.2.5 Exploiting the capabilities of remote FPGAs

In Section 2.3.1 we have provided a short introduction to Microsoft Catapult cloud data center architecture and how it integrates a pool of FPGAs [33]. Figure 7.12 recalls the architecture of Microsoft Catapult. In Microsoft Catapult, FPGAs are organized in a pool, and each FPGA can *donate* its unused resources to the pool. FPGAs that need them can then access to these resources trough a low latency connection (the order of tens of μs or less), based on Lightweight Transport Layer (LTL) protocol [34].

Currently *Slot* is applied to one or more applications which are executed onto a single FPGA. In future, we plan to apply it to better target architectures that allow

to exploit the capabilities of remote FPGAs. If we assume that this latency is very low with regards to the HET of tasks and with regards to the FPGA reconfiguration, a solution could simply be to adapt the current *Slot* by summing up together the resources of remote FPGAs.

7.2.6 Dynamic adaptation of scheduling in cloud

We would like to better adapt *Slot* in order to better tackle dynamic changes that the context of cloud data centers may require. In particular, we would like to dynamically re-run *Slot* everytime a new application is assigned to an FPGA which is already running an application. Let us consider the following example:

1. An application X is selected for scheduling onto an FPGA F
2. The orchestrator runs *Slot*: k slots are computed
3. Application X starts to execute at time t_0
4. At time $t_1 > t_0$, an application Y is assigned to the same FPGA F. Always at time t_1, application X has already completed the execution of the w slots, $w < k$.
5. At this point, we would like to automatically re-run *Slot* over a new application Z, which is an artificial application composed of an artificial source and sink that join the tasks of application Y with the tasks of the remaining $k - w$ slots of application X.
6. New slots are calculated and they are executed onto FPGA F
7. This would give the illusion, to the owners of applications, to be the only users of the FPGA F. Moreover, the owner of application Y would see its application immediately execute, without annoying waiting queues.

Appendix A

Low-level details of the MILP formulation

The MILP solver that we used for our experiments is the free and open source GLPK C library [7]. GLPK represents a LP or MILP problem instance as a constraint matrix $M \in \mathbb{R}^{n \times m}$. Each column j of M corresponds to one *structural variable* σ_j, that is, an input or output variable of the problem, and each row i corresponds to one *auxiliary variable* ξ_i used to express a constraint on the structural variables. The difference between LP and MILP is that in MILP structural variables can be constrained to be integer or binary values, while in LP all variables are real.

If we denote $\boldsymbol{\xi} \in \mathbb{R}^n$ the vector of auxiliary variables and $\boldsymbol{\sigma} \in \mathbb{R}^m$ the vector of structural variables, then:

$$\boldsymbol{\xi}^T = M \times \boldsymbol{\sigma}$$

Each structural or auxiliary variable v is also constrained by inequalities. Structural and auxiliary constraints can take one of five forms (L_v, U_v and V_v are some user-specified constant values):

- $-\infty \leq v \leq +\infty$ (unbounded)
- $L_v \leq v \leq +\infty$ (lower bound)
- $-\infty \leq v \leq U_v$ (upper bound)
- $L_v \leq v \leq U_v$ (double-bounded)
- $V_v \leq v \leq V_v$ (fixed)

The objective function is expressed as a linear combination of the structural variables, plus a constant shift, and a direction (minimize or maximize):

$$\min / \max \left(\lambda + \sum_{0 \leq j < m-1} \lambda_j \times \sigma_j \right)$$

Before we present our formulation we must explain why it uses more structural and auxiliary variables than strictly needed. The reader could indeed wonder why we uselessly added to the already great complexity of the MILP solving. There are always several ways to model a problem instance for MILP solving. A constant input value V, for instance, can be presented as an input structural variable σ, that is, a column of the constraint matrix M, with a fixed constraint:

$$V \leq \sigma \leq V$$

The V constant input value can then appear in the expression of an auxiliary variable ξ if its coefficient μ in the linear combination is not null and thus participate the auxiliary constraint:

$$L \leq \xi = \cdots + \mu \times \sigma + \cdots \leq U$$

But it can also not be presented as an input structural variable and appear directly in the bounds of the constraint:

$$\begin{aligned} L' &= L - \mu \times V \\ U' &= U - \mu \times V \\ L' &\leq \xi \leq U' \end{aligned}$$

In the formulation we designed the tasks durations are input structural variables because the coding was simpler and more regular but they could as well be constants used to compute the bounds of the auxiliary constraints.

A constant input value can even be used in the two forms and appear both as an input structural variable and in the bounds of constraints. This is the case with the horizon in the constraints that a task execution time is entirely contained in the execution time of the slot it is assigned to. In the following we thus denote $\mathcal{H}$ the constant value of the horizon and $\hbar$ the input structural variable constrained to have this value.

Another degree of freedom comes from some output variables that are indeed intermediate variables we are not really interested in. Task start and end times x_t

and z_t, for instance, are redundant. One can be computed from the other and the durations y_t. We can thus add the start and end times in the set of output structural variables if it is more convenient to express some constraints but we do not have to. We could also replace, for instance, the end time by $x_t + y_t$ in all linear combinations where they appear. And each time we add such optional output variables we usually also need to add extra auxiliary variables to express their relation with the other variables. For each task, for instance, we must add the following auxiliary variable and fixed constraint:

$$0 \leq a_t = x_t + y_t - z_t \leq 0$$

It is the same with the slot durations, start and end times u_s, v_s and w_s, three sets of outputs that could easily be reduced to two. This reduction would also avoid the following auxiliary variables:

$$0 \leq c_s = u_s + v_s - w_s \leq 0$$

All these non-essential variables add to the number of rows and columns of the constraint matrix M. And in most cases it is perfectly possible to get rid of them by hard-wiring the values of the inputs variables in the bounds of constraints, by replacing the intermediate output variables by their expressions and by removing the now useless auxiliary variables. One could argue that the larger the constraint matrix, the more difficult the solving and the longer the solver's run-time.

But in practice this is not the case because the difficulty of the MILP solving also strongly depends on the density and on the values of the elements of the constraint matrix. In order to assess this we tested several variants of the formulation, including one where most input variables have been transformed into constants and only essential output variables and auxiliary variables have been kept, leading to the smallest possible constraint matrix. This "optimized" version was the worst in terms of run-time with an average slow-down factor of 10 compared to the version we present here, which is the one that exhibits the best average run-times, even if it uses more variables than the minimum.

We already introduced the structural variables of our formulation in Section 5.2; they are listed in table A.1 where $|T_t|$ is the duration of task T_t. The variables with fixed constraints, like, for instance the task durations y_t, are inputs while the others are outputs to be computed by the solver. What makes this an MILP problem instead of an LP problem is only the constraint that the $\alpha_{t,s}$ task-to-slot allocation variables are binary.

The auxilliary variables and their bounds are listed in table A.2 where t_0 and t_1 are two task indexes such that $\pi_{t_0,t_1} = 1$ (task T_{t_0} precedes task T_{t_1}).

Name	Min	Max	Type	Definition
x_0	0	0	Real	Start time of task T_0
$x_{t>0}$	0	$\mathcal{H}$	Real	Start time of task $T_{t>0}$
y_t	$\lvert T_t \rvert$	$\lvert T_t \rvert$	Real	Duration of task T_t
z_t	0	$\mathcal{H}$	Real	End time of task T_t
u_1	0	0	Real	Start time of slot S_1
$u_{s>1}$	0	$\mathcal{H}$	Real	Start time of slot $S_{s>1}$
v_s	0	$\mathcal{H}$	Real	Duration of slot S_s
w_s	0	$\mathcal{H}$	Real	End time of slot S_s
$\alpha_{0,1}$	1	1	Bin.	Allocation of task T_0 to slot S_1
$\alpha_{0,s>1}$	0	0	Bin.	Allocation of task T_0 to slot S_1
$\alpha_{t>0,s}$	0	1	Bin.	Allocation of task $T_{t>0}$ to slot S_s
$\hbar$	$\mathcal{H}$	$\mathcal{H}$	Real	Horizon as a structural variable

Table A.1: Structural variables

Name	Min	Max	Definition	Expressed constraint
a_t	0	0	$x_t + y_t - z_t$	Task end time $=$ start time $+$ duration
$b_{s>1}$	$\mathcal{T}$	$\mathcal{T}$	$u_s - w_{s-1}$	Slot start time $=$ previous slot end time $+$ reconfiguration time
c_s	0	0	$u_s + v_s - w_s$	Slot end time $=$ start time $+$ duration
d_t	1	1	$\sum_{1 \leq s \leq n_s} \alpha_{t,s}$	Tasks are allocated to one single slot
$e_{s,r}$	0	q_r	$\sum_{0 \leq t < n_t} \alpha_{t,s} \times \rho_{t,r}$	No resource overuse
$f_{t,s}$	0	$2 \times \mathcal{H}$	$\hbar - \alpha_{t,s} \times \mathcal{H} + x_t - u_s$	Task start time $\geq$ its slot start time
$g_{t,s}$	0	$2 \times \mathcal{H}$	$\hbar - \alpha_{t,s} \times \mathcal{H} + w_s - z_t$	Task end time $\leq$ its slot end time
h_{t_0,t_1}	0	$\mathcal{H}$	$x_{t_1} - z_{t_0}$	Successor task start time $\geq$ its predecessor task end time

Table A.2: Auxiliary variables

The objective function is the end time of the T_{n_t-1} sink task and the objective consists in minimizing it:

$$\min(z_{n_t-1})$$

Appendix B

List of acronyms

List of recurring acronyms in this thesis:

- **ACO:** Ant Colony Optimization
- **ALM:** Adaptative Logic Module
- **ASIC:** Application Specific Integrated Circuit
- **AWS:** Amazon Web Services
- **BFS:** Breadth-First Search
- **BPP:** Bin-Packing Problem
- **BRAM:** Block-RAM
- **CLB:** Configurable Logic Element
- **CUT:** CPU User Time
- **DAG:** Direct Acyclic Graph
- **DRAM:** Dynamic RAM
- **DSE:** Design Space Exploration
- **DSP:** Digital Signal Processing
- **ECDF:** Empirical Cumulative Distribution Function
- **EDF:** Earliest Deadline First
- **EMB:** Embedded Memory Block

- **EC:** Elastic Cloud
- **FPGA:** Field Programmable Gate Array
- **GA:** Genetic Algorithm
- **GPP:** General Purpose Processor
- **GPU:** Graphics Processing Unit
- **HDL:** Hardware Description Language
- **HEFT:** Heterogeneous Earliest Finish Time
- **HEFT-NF:** Heterogeneous Earliest Finish Time Next-Fit
- **HET:** Hardware Execution Time
- **IP:** Intellectual Property
- **LE:** Logic Element
- **LUT:** Look-Up Table
- **MDE:** Model-Driven Engineering
- **MH:** Meta-Heuristic
- **MILP:** Mixed-Integer Linear Programming
- **NIC:** Network Interface Card
- **NRE:** Non-Recurring Engineering
- $n^{reconfig}$**:** number of reconfigurations
- **PLD:** Programmable Logic Device
- **RCSP:** Resource Constrained Scheduling Problem
- **RTOS:** Real Time Operating System
- **SA:** Simulating Annealing
- **SET:** Software Execution Time
- **SoC:** System on Chip
- **SRAM:** Static RAM

- **TS:** Tabu Search
- **TSET:** True Software Execution Time
- **TOR:** Top-Of-the-Rack
- T_R**:** Total Reconfiguration (time)
- T_∞**:** T infinite
- **WCET:** Worst Case Execution Time

Bibliography

[1] Altera - intel fpgas. https://www.intel.com/content/www/us/en/products/programmable.html.

[2] Altera Stratix V D5 - overview. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-v/stx5_51001.pdf. 2015.

[3] AWS - elastic block store. https://aws.amazon.com/ebs/.

[4] AWS - elastic compute cloud. https://docs.aws.amazon.com/ec2/index.html?nc2=h_ql_doc_ec2.

[5] AWS - kinesis. https://aws.amazon.com/kinesis/.

[6] AWS - simple storage service. https://aws.amazon.com/s3/.

[7] Glpk (gnu linear programming kit). https://www.gnu.org/software/glpk/.

[8] MARTE - modeling and analysis of real-time and embedded systems. http://www.omgmarte.org/. 2008.

[9] Microblaze soft processor core.

[10] Nios® ii processors for fpgas - intel® fpga.

[11] The resource-constrained project scheduling problem. http://www.iste.co.uk/data/doc_dtalmanhopmh.pdf.

[12] Ryft - powering elasticsearch in the cloud. https://www.xilinx.com/support/documentation/product-briefs/ryft-aws-f1.pdf. 2017.

[13] TTool. https://ttool.telecom-paris.fr/diplodocus.html. 2020.

[14] TTool - simulation trace of slot. https://github.com/SLOTAlgorithm-FPGA/SimulationTracesTTool. 2020.

[15] Xilinx. https://www.xilinx.com/.

[16] Xilinx Ultrascale - overview. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf. 2020.

[17] Xilinx virtex ultrascale+. https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html. 2020.

[18] Xilix 7-Series - overview. https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. 2018.

[19] *Deadline Scheduling for Real-Time Systems*. Springer US, 1998.

[20] Ieee standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, 2018.

[21] Ieee standard for vhdl language reference manual. *IEEE Std 1076-2019*, pages 1–673, 2019.

[22] Opencl, 2020.

[23] F. Abel, J. Weerasinghe, C. Hagleitner, B. Weiss, and S. Paredes. An fpga platform for hyperscalers. In *2017 IEEE 25th Annual Symposium on High-Performance Interconnects (HOTI)*, pages 29–32, 2017.

[24] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl. Reconos: An operating system approach for reconfigurable computing. *IEEE Micro*, 34(1):60–71, 2014.

[25] L. Apvrille, W. Muhammad, R. Ameur-Boulifa, S. Coudert, and R. Pacalet. A uml-based environment for system design space exploration. In *2006 13th IEEE International Conference on Electronics, Circuits and Systems*, pages 1272–1275, 2006.

[26] Hamid Arabnejad. List based task scheduling algorithms on heterogeneous systems-an overview. 2012.

[27] S. Banerjee, E. Bozorgzadeh, and N. Dutt. Exploiting application data-parallelism on dynamically reconfigurable architectures: Placement and architectural considerations. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(2):234–247, 2009.

[28] M. Bertolino, R. Pacalet, L. Apvrille, and A. Enrici. Efficient scheduling of fpgas for cloud data center infrastructures. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 57–64, 2020.

[29] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996.

[30] J. Blazewicz, J.K. Lenstra, and A.H Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1):11–24, 1983.

[31] M. Bogdanski, P. R. Lewis, T. Becker, and X. Yao. Improving scheduling techniques in heterogeneous systems with dynamic, on-line optimisations. In *2011 International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 496–501, 2011.

[32] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 3rd edition, 2011.

[33] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[34] Adrian Caulfield, Eric Chung, Doug Burger, and Derek Chiou. Lightweight transport protocol patent, December 2016.

[35] University of Tsukuba Center for Computational Sciences (CCS). Cygnus. `https://www.ccs.tsukuba.ac.jp/eng/supercomputers/#Cygnus`. 2018.

[36] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling fpgas in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, New York, NY, USA, 2014. Association for Computing Machinery.

[37] Katherine Compton and Scott Hauck. Reconfigurable computing: A survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, June 2002.

[38] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[39] J. Cui, X. Chen, Y. Lei, and W. Xu. Improving the efficiency of scheduling and placement in fpga by small-world model based genetic algorithm. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 99–106, 2010.

[40] K. Danne, Roland Mühlenbernd, and Marco Platzner. Server-based execution of periodic tasks on dynamically reconfigurable hardware. *Computers and Digital Techniques, IET*, 1:295–302, 08 2007.

[41] K. Danne and M. Platzner. Periodic real-time scheduling for fpga computers. In *Third International Workshop on Intelligent Solutions in Embedded Systems, 2005.*, pages 117–127, 2005.

[42] K. Danne and M. Platzner. Partitioned scheduling of periodic real-time tasks onto reconfigurable hardware. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pages 8 pp.–, 2006.

[43] E. A. Deiana, M. Rabozzi, R. Cattaneo, and M. D. Santambrogio. A multiobjective reconfiguration-aware scheduler for fpga-based heterogeneous architectures. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, 2015.

[44] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996.

[45] K. Du, S. Domas, and M. Lenczner. A solution to overcome some limitations of sdf based models. In *2018 IEEE International Conference on Industrial Technology (ICIT)*, pages 1395–1400, 2018.

[46] K. Du, S. Domas, and M. Lenczner. Actors with stretchable access patterns. *Integration*, 66:44–59, 2019.

[47] Ke Du. *Building and analyzing processing graphs on FPGAs with strong time and hardware constraints*. Theses, Université Bourgogne Franche-Comté, April 2018.

[48] Andrea Enrici, Ludovic Apvrille, and Renaud Pacalet. A model-driven engineering methodology to design parallel and distributed embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 22(2):34:1–34:25, January 2017.

[49] F. Ferrandi, P. L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo. Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(6):911–924, 2010.

[50] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications. pages 47–56, 02 2012.

[51] W. Fu and K. Compton. Scheduling intervals for reconfigurable computing. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 87–96, 2008.

[52] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., USA, 1979.

[53] A. Gharbi, A. Enrici, B. Uscumlic, L. Apvrille, and R. Pacalet. Efficient and exact design space exploration for heterogeneous and multi-bus platforms. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 16–23, 2020.

[54] A. Ghosal, R. Limaye, K. Ravindran, S. Tripakis, A. Prasad, G. Wang, T. N. Tran, and H. Andrade. Static dataflow with access patterns: Semantics and analysis. In *DAC Design Automation Conference 2012*, pages 656–663, 2012.

[55] M. Gotz, F. Dittmann, and C. E. Pereira. Deterministic mechanism for run-time reconfiguration activities in an rtos. In *2006 4th IEEE International Conference on Industrial Informatics*, pages 693–698, 2006.

[56] Alexandre Graell i Amat and Ragnar Thobaben. Chapter 9 - an introduction to distributed channel coding. In David Declerq, Marc Fossorier, and Ezio Biglieri, editors, *Academic Press Library in Mobile and Wireless Communications*, pages 399 – 450. Academic Press, Oxford, 2014.

[57] J.L. Gross, J. Yellen, and M. Anderson. *Graph Theory and its Applications*. CRC Press, 3rd edition, 2018.

[58] Marcelo Götz, Achim Rettberg, Carlos Pereira, and Franz Rammig. Run-time reconfigurable rtos for reconfigurable systems-on-chip. *J. Embedded Computing*, 3:39–51, 01 2009.

[59] Sarah Harris and David Harris. *Digital Design and Computer Architecture: ARM Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2015.

[60] Sönke Hartmann and Dirk Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem. Working Paper 02, 2008.

[61] Scott Hauck. Configuration prefetch for single context reconfigurable coprocessors. 03 2000.

[62] M. Huang, H. Simmler, O. Serres, and T. El-Ghazawi. Rdms: A hardware task scheduling algorithm for reconfigurable computing. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, 2009.

[63] Miaoqing Huang, Vikram Narayana, Harald Simmler, Olivier Serres, and Tarek El-Ghazawi. Reconfiguration and communication-aware task scheduling for high-performance reconfigurable computing. *TRETS*, 3:20, 11 2010.

[64] Miaoqing Huang, Harald Simmler, Proshanta Saha, and Tarek El-Ghazawi. Hardware task scheduling optimizations for reconfigurable computing. pages 1 – 10, 12 2008.

[65] Reakook Hwang, Mitsuo Gen, and Hiroshi Katayama. A comparison of multiprocessor task scheduling algorithms with communication costs. *Computers and Operations Research*, 35(3):976 – 993, 2008. Part Special Issue: New Trends in Locational Analysis.

[66] R. Höller, D. Haselberger, D. Ballek, P. Rössler, M. Krapfenbauer, and M. Linauer. Open-source risc-v processor ip cores for fpgas — overview and evaluation. In *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–6, 2019.

[67] Peter Jamieson. Revisiting genetic algorithms for the fpga placement problem. pages 16–22, 01 2010.

[68] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Song Yao, Song Han, Yu Wang, and Huazhong Yang. From model to fpga: Software-hardware co-design for efficient neural network acceleration. In *2016 IEEE Hot Chips 28 Symposium (HCS)*, pages 1–27, 2016.

[69] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer-Verlag Berlin Heidelberg, 2004.

[70] Bart Kienhuis, Ed F. Deprettere, Pieter van der Wolf, and Kees Vissers. *A Methodology to Design Programmable Embedded Systems*, pages 18–37. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[71] Steve Kilts. *Advanced FPGA Design*. John Wiley & Sons, Ltd, 2007.

[72] J. Kleinberg and E Tardos. *Algorithm Design*. Pearson/Addison-Wesley, Boston, MA, 2005.

[73] D. Knorreck. *UML-Based Design Space Exploration, Fast Simulation and Static Analysis*. PhD thesis, Telecom Paris, 2011.

[74] Daniel Knorreck, Ludovic Apvrille, and Renaud Pacalet. Formal system-level design space exploration. *Concurrency and Computation: Practice and Experience*, 25(2):250–264, 2013.

[75] Bernhard Korte and Jens Vygen. *Combinatorial Optimization - Theory and Algorithms*. Springer-Verlag Berlin Heidelberg, 6th edition, 2018.

[76] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[77] Edward Lee and David Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *Computers, IEEE Transactions on*, C-36:24 – 35, 02 1987.

[78] L. W. Li, L. Apvrille, and D. Genius. Virtual prototyping of automotive systems : Towards multi-level design space exploration. 2016.

[79] C. Y. Lin, N. Wong, and H. K. So. Operation scheduling for fpga-based reconfigurable computers. In *2009 International Conference on Field Programmable Logic and Applications*, pages 481–484, 2009.

[80] Andrea Lodi, Silvano Martello, and Michele Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241–252, 2002.

[81] Andrea Lodi, Silvano Martello, and Daniele Vigo. Recent advances on two-dimensional bin packing problems. *Discrete Applied Mathematics*, 123(1):379–396, 2002.

[82] Y. Lu, T. Marconi, K. Bertels, and G. Gaydadjiev. A communication aware online task scheduling algorithm for fpga-based partially reconfigurable systems. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 65–68, 2010.

[83] Y. Lu, T. Marconi, G. Gaydadjiev, and K. Bertels. An efficient algorithm for free resources management on the fpga. In *2008 Design, Automation and Test in Europe*, pages 1095–1098, 2008.

[84] K. F. Man, K. S. Tang, and S. Kwong. Genetic algorithms: concepts and applications [in engineering design]. *IEEE Transactions on Industrial Electronics*, 43(5):519–534, 1996.

[85] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., USA, 1990.

[86] D. Merkle, M. Middendorf, and H. Schmeck. Ant colony optimization for resource-constrained project scheduling. *IEEE Transactions on Evolutionary Computation*, 6(4):333–346, 2002.

[87] Orlando Moreira and Marco Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007, 01 2007.

[88] Alvaro Neuenfeldt Júnior. *The Two-Dimensional Rectangular Strip Packing Problem*. PhD thesis, 12 2017.

[89] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric Chung. Accelerating deep convolutional neural networks using specialized hardware, February 2015.

[90] Peter Poplavko, Twan Basten, Marco Bekooij, Jef Meerbergen, and B. Mesman. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. pages 63–72, 01 2003.

[91] A. Putnam. Large-scale reconfigurable computing in a microsoft datacenter. In *2014 IEEE Hot Chips 26 Symposium (HCS)*, pages 1–38, 2014.

[92] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*, 35(3):10–22, 2015.

[93] Yang Qu, Juha-Pekka Soininen, and Jari Nurmi. Static scheduling techniques for dependent tasks on dynamically reconfigurable devices. *Journal of Systems Architecture*, 53:861–876, 11 2007.

[94] Imran Quadri, Abdoulaye Gamatié, Samy Meftali, jean-luc Dekeyser, Huafeng Yu, and Éric Rutten. Targeting reconfigurable fpga based socs using the marte uml profile: from high abstraction levels to code generation. *International Journal of Embedded Systems*, 4, 09 2010.

[95] Imran Quadri, Samy Meftali, and jean-luc Dekeyser. High level modeling of dynamic reconfigurable fpgas. *International Journal of Reconfigurable Computing*, 2009, 05 2009.

[96] Reza Ramezani. A prefetch-aware scheduling for fpga-based multi-task graph systems. *The Journal of Supercomputing*, 01 2020.

[97] B. Ringlein, F. Abel, A. Ditter, B. Weiss, C. Hagleitner, and D. Fey. System architecture for network-attached fpgas in the cloud using partial reconfiguration. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 293–300, 2019.

[98] M. Santambrogio. Fpga computing systems - coursera. https://www.coursera.org/learn/fpga-intro.

[99] Valentin Savin. Chapter 4 - ldpc decoders. In David Declerq, Marc Fossorier, and Ezio Biglieri, editors, *Academic Press Library in Mobile and Wireless Communications*, pages 211 – 259. Academic Press, Oxford, 2014.

[100] Yang Shi, Zhaoyun Chen, Wei Quan, and Mei Wen. A Performance Study of Static Task Scheduling Heuristics on Cloud-Scale Acceleration Architecture. In *ICCDE*, pages 81–85, 2019.

[101] D. Sisejkovic. Evolution of scheduling heuristics for the resource constrained scheduling problem. https://www.semanticscholar.org/paper/EVOLUTION-OF-SCHEDULING-HEURISTICS-FOR-THE-RESOURCE-Sisejkovic/c43fd94c72e7342b94f0cd025fa861f3d923c81f, 2016.

[102] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1393–1407, 2004.

[103] INRIA DaRT team. GASPARD - soc framework. http://www.gaspard2.org/. 2009.

[104] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. on Par. and Dist. Sys.*, 13(3):260–274, 2002.

[105] Wim Vanderbauwhede and Khaled Benkrid, editors. *High-Performance Computing Using FPGAs*. Springer New York, 2013.

[106] G. Wassi, Mohamed El Amine Benkhelifa, G. Lawday, F. Verdier, and S. Garcia. Multi-shape tasks scheduling for online multitasking on fpgas. In *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–7, 2014.

[107] Vincent M. Weaver. List based task scheduling algorithms on heterogeneous systems-an overview. In *2013 FastPath Workshop*, 2013.

[108] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *2007 44th ACM/IEEE Design Automation Conference*, pages 658–663, 2007.

[109] Xiao Fan Wang and Guanrong Chen. Complex networks: small-world, scale-free and beyond. *IEEE Circuits and Systems Magazine*, 3(1):6–20, 2003.

[110] Yang Qu, J. . Soininen, and J. Nurmi. A parallel configuration model for reducing the run-time reconfiguration overhead. In *Proceedings of the Design Automation Test in Europe Conference*, volume 1, pages 1–6, 2006.

[111] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. pages 10–10, 06 2012.

[112] Y. Zhao, C. Tian, Z. Zhu, J. Cheng, C. Qiao, and A. X. Liu. Minimize the make-span of batched requests for fpga pooling in cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 29(11):2514–2527, 2018.

Titre : Ordonnancement Efficace des Applications sur Cloud FPGAs

Mots clés : algorithmes, ordonnancement, cloud, FPGAs, architectures

Résumé : Cette thèse a été faite à Télécom Paris et a été financée par Nokia Bell Labs France. Les contributions de la thèse trouvent leur motivation d'une part dans l'usage croissant d'accélérateurs matériel, comme les FPGAs pour décharger les processeurs de fonctions complexes. D'autre part, ces accélérateurs sont de plus en plus intégrés dans des centre de calculs (”cloud”). Ces FPGAs peuvent être loués aux utilisateurs, et leur coût d'utilisation dépend alors de la durée allouée. Ainsi, il devient intéressant de mettre en place des techniques permettant de minimiser le temps d'exécution des applications.
Dans cette thèse, nous considérons des applications ”flots de données”, c'est à dire qu'une tâche attend des données d'une tâche précédente avant de s'exécuter, et produit des données en sortie qui sont utilisées en entrée d'autres tâches. Nous prenons aussi comme hypothèse que ces applications sont pré-caractérisées par leur temps d'exécution sur FPGA ainsi que par les éléments matériels dont elles ont un besoin exclusif sur le FPGA (par exemple : les éléments logiques).
Il existe déjà des travaux pour choisir un bon ordonnancement de tâches sur FPGAs. Malheureusement, ces derniers utilisent une recherche exhaustive de la meilleur solution, ce qui est très coûteux en temps de calcul, soit ils reposent sur des heuristiques rapides mais dont la solution produite en en règle générale assez éloignée de la solution optimale. De nombreux travaux considèrent aussi des modèles de FPGA trop abstraits : par exemple, le FPGA est modélisé uniquement sous la forme d'un ensemble d'éléments logiques, omettant par exemple les DSPs intégrés aux FPGAs, rendant ainsi la solution calculée éventuellement non utilisable sur un FPGA (concret).
La solution que nous proposons dans notre thèse repose sur une modélisation plus réaliste des FPGAs et sur une nouvelle heuristique d'ordonnancement. Cette dernière est itérative sur le graphe : elle extrait du graphe de tâches la tâche les plus complexes en termes de temps de calcul, puis utilise un système d'évaluation pour identifier les tâches les plus pertinentes à grouper avec la tâche la plus complexe. Notre évaluation effectuée sur un grand nombre de graphes d'applications fictives montre que notre heuristique est tout aussi rapide qu'une autre heuristique de référence, tout en offrant de meilleurs résultats. De plus, notre contribution a été intégrée à un environnement orienté modèles pour la conception de systèmes embarqués. De plus, nous dressons des perspectives intéressants pour l'utilisation de notre heuristique dans d'autres contextes.

Title : Efficient Scheduling of Applications onto Cloud FPGAs

Keywords : algorithms, scheduling, cloud, FPGAs, architectures

Abstract : This thesis has been realized in Télécom Paris and it has been financed by Nokia Bell Labs France. It founds its motivations in the increasing usage of hardware accelerators such as FPGAs and their recent integration in modern cloud data center. In some cases, servers and FPGAs are rented to users and the cost is related to the utilization time. Thus, offering a better sharing of FPGA pools would interest all stakeholders, namely cloud providers and users. We focus on scheduling and, in particular, we focus on makespan minimization of applications. The latter are assumed to be composed of several dependent tasks, whose features (i.e., dependencies, execution time, resource requirements, and so on) are known prior to their execution. With respect to the state of the art, we have sought to design an approach which is, at the same time, (i) general, (ii) fast and (iii) of high-quality. Indeed, several related works represent the applications and the architecture through simple models (e.g., the FPGA is often represented only with the amount of reconfigurable logic). We retain that such simple models may lead to unfeasible scheduling. Moreover, the vast majority of them is either based on slow and precise algorithms or on fast heuristics whose quality is far from the optimum. We therefore propose a scheduling solution characterized by a good quality in terms of makespan while keeping the decision time in the order of tens of milliseconds for common applications. The main contributions of the thesis are a modelling proposal for FPGAs, the design of a heuristic which targets the makespan minimization and the evaluation of this heuristic on a synthetic benchmark of pseudo-randomly generated applications. Additionally, we have integrated this method to a model-driven engineering (MDE) tool to better support the early design of embedded systems. Finally, we propose several extensions to extend the approach to different architectures.

Institut Polytechnique de Paris
91120 Palaiseau, France