Definition and evaluation of spatio-temporal scheduling strategies for 3D multi-core heterogeneous architectures

Quang Hai Khuat

ANNÉE 2015

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Traitement du Signal et Télécommunications

École doctorale MATISSE

présentée par

Quang-Hai Khuat

préparée à l'unité de recherche UMR6074 IRISA
Institut de recherche en informatique et systèmes aléatoires - CAIRN
École Nationale Supérieure des Sciences Appliquées et de Technologie

---

Definition and evaluation of spatio-temporal scheduling strategies for 3D multi-core heterogeneous architectures

Thèse soutenue à Lannion le 10 Mars 2015
devant le jury composé de :

Daniel CHILLET
Maître de Conférences - HDR
Université de Rennes 1 / directeur de thèse

Michael HUEBNER
Professeur, Ruhr-University Bochum / examinateur

Sebastien PILLEMENT
Professeur, Université de Nantes / examinateur

Jean-Philippe DIGUET
Directeur de recherche CNRS
Université de Bretagne-Sud / rapporteur

Fabrice MULLER
Maître de Conférences - HDR
Université de Nice-Sophia Antipolis / rapporteur

Benoît MIRAMOND
Maître de Conférences - HDR
Université de Cergy-Pontoise / examinateur

Emmanuel CASSEAU
Professeur, Université de Rennes 1 / invité

Acknowledgements

First of all, I would like to express the sincere thanks to my supervisor Dr. Hab. Daniel Chillet for his support throughout my PhD work. Without his constructive suggestions, his guidance, his cheerful enthusiasm and his moral support, it would not have been possible to finish this PhD thesis.

I grateful thank Prof. Dr. Michael Hübner for hosting me in ESIT team for my three months of mobility. His positive remarks, his advices and his kindness have always been helpful and invaluable for my research.

I would like to thank all the members of the jury who evaluated this work: Jean-Phillipe Diguet (Director of Research CNRS at Labsticc, Université de Bretagne-Sud, Lorient), Fabrice Muller (Dr. Hab. at Université de Nice-Sophia Antipolis), Sebastien Pillement (Professor at Université de Nantes) and Benoît Miramond (Dr. Hab. at Université de Cergy-Pontoise). I am equally thankful to Prof. Dr. Emmanuel Casseau for accepting to attend my PhD defense as invited professor.

I wish to thank Prof. Dr. Olivier Sentieys, the head of CAIRN, for welcoming me in the team. It is an honor for me to work and complete the PhD thesis in such a dynamic team. Also, I want to thank all the members of CAIRN for making my three years of PhD a memorial period.

Finally, I would like to thank my parents, my parents-in-law, my brother for the support they provide me through my entire life and in particular, I must acknowledge my wife, without whose love, encouragement and editing assistance, I would not have finished this thesis.

Contents

List of Figures

List of Tables

Context

Les systèmes embarqués sont présents partout dans notre vie quotidienne. Nous observons qu'ils sont intégrés dans une grande variété de produits : téléphone portable, machine à laver, voiture, avion, équipements médicaux, etc. Une grande majorité des microprocesseurs fabriqués de nos jours sont consacrés aux plate-formes de type systèmes embarqués. Un système embarqué peut contenir un ou plusieurs processeurs associés à d'autres composants comme les mémoires, les périphériques, les bus de communications et des composants spécifiques dédiés pour les applications cibles (comme le DSP, les accélérateurs, etc). L'évolution des technologies de fabrication conduit, d'année en année, à des composants de plus en plus petite taille et offrant des performances toujours plus importantes. Ces composants de systèmes embarqués peuvent donc être intégrés dans une seule puce, conduisant à ce qu'on appelle le système sur puce, ou encore "System On Chip" (SoC) en anglais.

Parallèlement à cette évolution des systèmes embarqués, les applications d'aujourd'hui sont de plus en plus complexes et gourmandes en puissance de calcul, en mémoire, en communication, etc. Les systèmes multiprocesseurs sur puce (MPSoC) sont des solutions qui peuvent répondre à cette complexité. Ces systèmes offrent non seulement une certaine flexibilité, grâce à la reprogrammation logicielle, mais aussi une grande capacité à exécuter en parallèle de nombreuses fonctionnalités. Ces solutions résultent de hautes performances pour le système mais ont l'inconvénient d'être statiques, i.e. elles ne permettent pas une adaptation et/ou des modifications après leur fabrication pour pouvoir s'adapter aux dynamismes d'applications.

Afin de répondre favorablement aux dynamismes des applications, les MPSoC doivent intégrer des ressources matérielles reconfigurables. Cela est rendu possible par l'intégration de zones matérielles reconfigurables de type FPGA. Ces zones apportent la capacité d'adaptation et de reconfiguration au circuit tout en offrant des niveaux de performances très élevés. Au lieu de développer des circuits intégrés spécifiques à une application (ASIC), ce qui nécessite un délai de conception et un coût de production importants, le fait d'utiliser des zones reconfigurables, FPGA, donne la possibilité de mettre en oeuvre un nouveau système en reconfigurant les fonctionnalités adaptées aux nouvelles applications. De plus, un des avantages de l'utilisation d'un FPGA au sein d'un MPSoC est la reconfiguration dynamique et partielle. Cette capacité permet de reconfigurer une partie

du FPGA en temps réel sans interrompre les autres parties en cours d'exécution. Le système combinant MPSoC et FPGA est appelé Reconfigurable Multiprocessor System On Chip (MPRSoC). Ce système disposé d'un support d'exécution logicielle et matérielle offert à la fois la haute performance, la flexibilité tout en limitant la surface globale du système.

L'évolution des systèmes sur puce ne cesse de s'accélérer, depuis environ une cinquantaine d'années. Aujourd'hui, cette évolution se poursuit avec l'apparition des technologies dites 3D, permettant la conception de systèmes sur puce en trois dimensions (ou 3DSoC). Comparer aux SoC planaires, cette technologie permet d'empiler verticalement des puces les unes sur les autres pour former un circuit en "stack". Il en résulte une augmentation des performances, une réduction de la longueur de communication en remplaçant la connexion horizontale par une courte connexion verticale, une réduction du coût de production en choisissant la technologie adaptée pour chaque puce et finalement une réduction de facteur de forme.

Les architectures considérées dans mon travail de thèse disposent de capacités de reconfiguration, il s'agit de circuits dit Reconfigurable System on Chip en trois dimensions (3DRSoC). Ces plate-formes sont constituées de deux couches qui sont verticalement connectées : la couche multiprocesseurs et la couche reconfigurable. Le fait d'empiler ces deux couches verticalement permet de conserver les caractéristiques du RSoC planaire tout en héritant des avantages offerts par la technologie 3D. En effet, en utilisant les connections verticales (de type microbumps ou TSVs), les communications entre la partie logicielle sur le MPSoC et la partie matérielle sur la zone reconfigurable peuvent être plus rapides et mieux assurées. Par conséquent, les architectures 3DRSoC sont une solution prometteuse qui répond mieux aux plus grandes variétés d'applications.

Motivations et Objectives

Le traitement d'une application est souvent découpé en tâches avec les dépendances entre elles. À cause de la complexité des applications, chaque tâche peut avoir différentes implémentations logicielles et/ou matérielles. Ces implémentations donnent la possibilité, pour les tâches, d'être exécutées sur les différents composants de l'architecture. L'implémentation logicielle de la tâche (ou tâche logicielle) est une portion de code exécutable

sur un processeur. L'implémentation matérielle de la tâche (ou tâche matérielle) est une fonction synthétisée et configurable dans le FPGA. La gestion globale de l'architecture 3DRSoC nécessite un système d'exploitation adapté (Operating System OS) qui consiste à organiser l'ensemble des traitements d'une application sur cette plate-forme. Parallèlement aux services de communication ou de gestion mémoire, cet OS doit également fournir des méthodes d'ordonnancement pour la gestion et l'utilisation efficace des ressources de calcul. Lors de l'exécution d'une application sur une architecture 3DRSoC, ces méthodes devront être capables de déterminer les ressources (le processeur ou la zone du FPGA) qui vont être utilisées par chaque tâche (dimension spatiale) à un instant donné (dimension temporelle) pour satisfaire les contraintes de coût de communication, de puissance de calcul, de consommation d'énergie, de temps d'exécution, etc. On parlera donc de l'ordonnancement spatio-temporel.

Pour les applications n'ayant pas de dynamisme, les décisions spatio-temporelles peuvent être prises hors-ligne, i.e. avant que l'application commence son exécution sur la plate-forme. Dans ce cas là, nous pouvons assurer l'optimalité des décisions. Cependant, pour les applications dynamiques dont le comportement dépend des événements extérieurs, les décisions spatio-temporelles pour les tâches doivent être prises "en-ligne", i.e. pendant l'exécution. Dans ce cas, le flot d'exécution des tâches, ainsi que le support d'exécution pour chaque tâche ne sont pas connus a priori. À cause de cette caractéristique "en-ligne", nous ne pouvons pas garantir que ces décisions donneront la solution optimale mais plutôt une solution qui est "proche de l'optimum".

Dans ce travail de thèse, notre objectif est de proposer des stratégies d'ordonnancement spatio-temporel pour les architectures de type 3DRSoC. Nos stratégies ciblent deux objectifs : la minimisation du coût de communication entre les tâches et la minimisation du temps d'exécution global de l'application.

- **Minimisation du coût de communication entre les tâches** : Bien qu'il existe différents algorithmes d'ordonnancement spatio-temporel pour des RSoC planaire, la prise en compte de la troisième dimension de 3DRSoC rend le problème d'ordonnancement plus difficile à résoudre et ce problème se complexifie encore lorsque l'on considère les communications entre tâches. Comme les tâches peuvent être exécutées en logicielles et/ou en matérielles, elles peuvent communiquer de façon horizontale et/ou verticale. La communication entre deux tâches est liée aux temps

de transfert entre tâches et donc liée au nombre de données propagées lors de la communication. Plus la distance entre les tâches qui communiquent est grande, plus le coût de communication sera pénalisé. De plus, l'interconnexion de deux tâches affecte significativement le temps d'exécution global de l'application, la charge moyenne du réseau de communication, ainsi que la puissance et l'énergie consommées. Pour cette raison, il est très important de proposer des stratégies d'ordonnancement qui prennent en considération l'emplacement des tâches sur les trois dimensions afin de réduire au maximum le coût de communication entre les tâches.

- **Minimisation du temps d'exécution global de l'application** : Parce que les ressources du FPGA du 3DRSoC sont limitées, nous ne pouvons pas exécuter toutes les tâches de l'application en matérielle. Le support logiciel MPSoC du 3DRSoC est donc exploité pour offrir la possibilité d'exécuter certaines tâches en logicielle. Pendant l'exécution de l'application, les tâches sont anticipées sur une ressource de type processeur pour débuter leur traitement et libérées quand leur traitement est fini. À l'inverse d'une tâche logicielle dont l'exécution se fait sur un processeur, ces allocations et libérations des tâches matérielles peuvent causer la fragmentation du FPGA. Cela peut conduire à des situations indésirables où les futures tâches ne peuvent pas être placées sur le FPGA à cause de mauvais placements de tâches précédentes, même s'il y a suffisamment de zones libres. Ces tâches doivent attendre jusqu'au moment où il y aura des régions disponibles sur le FPGA pour les accueillir. Par conséquent, le temps d'exécution global de l'application sera augmenté et la performance globale du système sera pénalisée. Au lieu d'attendre que la tâche soit exécutée sur le FPGA, elle aurait pu être exécutée sur un des processeurs disponibles en vue d'anticiper leur traitement et ainsi s'achever plus rapidement. Nous nous intéressons aux stratégies d'ordonnancement spatio-temporel permettant de décider "en-ligne" des choix à prendre entre l'exécution logicielle et matérielle, à quel moment, sur quelle zone ou quel processeur pour minimiser le temps d'exécution global de l'application.

Contributions

Dans une architecture de type 3DRSoC, l'existence d'un FPGA est cruciale pour accélérer les traitements des tâches tout en maintenant une communication aisée et rapide avec les composants de la couche MPSoC. Rendre l'utilisation du FPGA plus efficace est extrêmement important pour atteindre la meilleure performance du 3DRSoC global.

Dans ce travail, deux types d'architectures 3DRSoC sont considérés : le 3DRSoC homogène et 3DRSoC hétérogène. La différence de ces deux architectures vient de différents types de FPGA. Dans le 3DRSoC homogène, le FPGA est un modèle contenant des ressources reconfigurables homogènes tandis que dans la 3DRSoC hétérogène, le FPGA est un modèle contenant des ressources reconfigurables hétérogènes.

Notre contribution, dans un premier temps, consiste à étudier les stratégies d'ordonnancement spatio-temporel "en-ligne" pour un ensemble de tâches matérielles s'exécutant sur les différentes types de FPGA. Nous proposons :

- **Pfair Extension for Reconfigurable Resource (Pfair-ERR)** qui est une stratégie d'ordonnancement spatio-temporel pour un FPGA de type 2D Bloc Area. Dans ce type d'architecture, le FPGA contient plusieurs zones reconfigurables qui sont prédéfinies et figées. Ordonnancer les tâches sur ce type d'architecture est équivalent à un ordonnancement sur un système multiprocesseur. Pfair-ERR est une extension d'un algorithme dit Pfair qui est considéré comme optimal pour maximiser l'utilisation des processeurs dans un système multiprocesseurs. Le but de Pfair-ERR est de modifier le Pfair classique pour prendre en compte les dépendances entre les tâches et minimiser le coût de communication entre elles tout en maximisant l'utilisation des ressources du FPGA. Le travail sur Pfair-ERR a été publié dans [3].
- **Vertex List Structure Best Communication Fit (VLS-BCF)** qui est une stratégie d'ordonnancement spatio-temporel pour un FPGA de type 2D Free Area. Ce type d'architecture contient des blocs logiques reconfigurables, les zones reconfigurables ne sont pas prédéfinies a priori mais adaptative par rapport à la taille (ou les ressources) demandée par les tâches. VLS-BCF est basé sur un algorithme dit Vertex List Structure (VLS) ayant une faible complexité et une simple structure de données pour la gestion des ressources de ce type de FPGA. L'objectif de

VLS-BCF est d'éviter de longues et couteuses communications, donc de réduire le coût de communication entre les tâches. Nous montrons ainsi que VLS-BCF permet de réduire également la probabilité de créer des points chauds sur le FPGA. Pour évaluer le VLS-BCF en termes de "points chauds", nous développons une solution dite VLS-BHF dont l'objective est d'ordonnancer et placer des tâches de façon à minimiser le nombre de "points chauds" tout en gardant un coût de communication faible. Ce travail a été publié dans [5].

- **Spatio-Temporal Scheduling strategy for Heterogeneous FPGA (STSH)** qui est une stratégie d'ordonnancement spatio-temporel pour un FPGA 2D hétérogène. Ce type d'architecture contient non seulement des blocs logiques reconfigurables mais aussi d'autres blocs hétérogènes. Cette hétérogénéité impose des contraintes de placement strictes pour les tâches et nécessite une stratégie de placement différente. STSH prend en considération cette hétérogénéité dans son placement. STSH combine la technique du "prefetching" avec deux autres facteurs : la priorité des tâches et le placement intelligent pour minimiser le temps d'exécution global de l'application. Le travail sur STSH a été publié dans [6].

Une fois que les stratégies d'ordonnancement sur les différentes architectures de FPGA ont été étudiées, nous étendons ces stratégies pour adresser le principal objectif de ces travaux qui a été de proposer des stratégies d'ordonnancement spatio-temporel "en-ligne" pour les architectures 3DRSoCs. Dans ce contexte, une tâche peut être exécutée matériellement et/ou logiciellement. Nous proposons les stratégies suivantes :

- **3D Spatio-Temporal Scheduling (3DSTS)** qui consiste à prendre en considération la troisième dimension pendant l'ordonnancement pour minimiser le coût de communication entre les tâches. La plate-forme considérée est une architecture 3DRSoC homogène dont la couche FPGA est de type 2D Bloc Area. 3DSTS évalue, pendant l'exécution de l'application, la nécessité de communiquer en face à face via les connections verticales pour trouver la meilleure instanciation possible des tâches (logicielle et matérielle) afin de minimiser le coût global de communication. Ce travail a été publié dans [7].

- **3D Hardware/Software with Software execution Prediction (3DHSSP)** qui est une stratégie d'ordonnancement spatio-temporel pour une architecture 3DR-SoC hétérogène dont le FPGA est un FPGA 2D hétérogène. L'objectif de 3DHSSP est de décider et d'évaluer, pendant l'exécution de l'application, quelle tâche est exécutée en logiciel ou quelle tâche est exécutée en matériel afin de minimiser le temps d'exécution global de l'application. 3DHSSP évalue l'intérêt de continuer l'exécution logicielle d'une tâche en cours, ou d'annuler ce traitement pour commencer, à partir de l'état initial, l'exécution matérielle de cette tâche. Ce travail a été publié dans [8].

Chapter 1

Introduction

Embedded systems are now present everywhere in our daily life and we observe their integration into a wide variety of products such as watches, cell phones, washing machines, cars, planes, medical equipments, etc. To support the execution of large number of applications, these embedded systems contain one or more processors associated with other components such as memories, peripherals, communications bus and specific components (such as DSP, accelerators, etc.). This evolution is supported by the sophisticated IC fabrication technology which makes the components becoming smaller and smaller over time while offering even greater performance. These embedded system components can be integrated into a single chip, leading to a system called System On Chip (SoC).

In parallel with the development of embedded systems, today's applications are more and more complex and intensive in power computing, memory and communication, etc. To solve the application complexity, Multiprocessor system on chip (MPSoC) appears as an interesting solution by offering not only a certain flexibility through the software reprogramming, but also a great ability to run in parallel many tasks. Using such system results a high performance for the system but the drawback is its "static" nature, i.e. it does not allow adaptations and/or modifications after its manufacturing to follow dynamic applications.

To address favorably the dynamism of applications, MPSoC often embeds reconfigurable hardware resources. This incorporation is totally possible with the integration of hardware reconfigurable circuits as Field-programmable gate array (FPGA). These circuits

provide the ability to adapt and reconfigure themselves while offering very high performance levels. Instead of developing a specific integrated circuits (ASIC), which requires a significant design time and an important manufactory cost, using reconfigurable circuits as FPGA gives the possibility to implement a new system by reconfiguring the features adapted to new applications. In addition, one advantage of using a FPGA within a MPSoC is the dynamic and partial reconfiguration paradigm. This capacity allows to reconfigure a portion of logic blocks during runtime without interrupting the rest of the system. Therefore, the system can change its behavior during runtime according to its environment or external events. The system combining MPSoC and FPGA is called Multiprocessor Reconfigurable System On Chip (MPRSoC). This system disposing of a software and hardware execution support offers both the high performance and the flexibility while reducing the global area of the system.

1.1 3D multicore heterogeneous architecture context

The evolution of SoC has been increased for about fifty years. Today, it continues with the emergence of a so-called 3D technology, enabling the design of SoCs in three dimensions (3DSoC). Compare with planar SoCs, this 3D technology allows stacking layers vertically on top of each other to form a circuit said "in stack". The expected results are an increase in performance, a reduction of communication wires by replacing the horizontal connections with a short vertical connections and a form factor reduction. Moreover, the manufacturing cost is also reduced as each layer can be fabricated and optimized using their respective technology before assembling to form a circuit in stack.

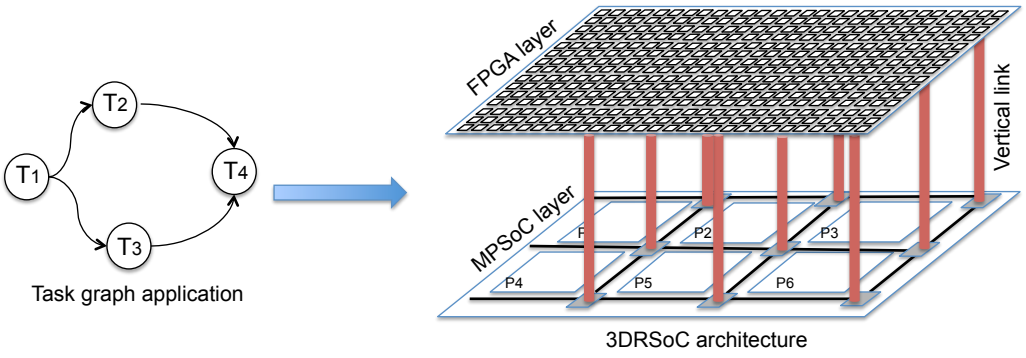


FIGURE 1.1: Mapping a task graph application on a 3DRSoC platform

The 3DSoCs having the reconfiguration capabilities is called 3D Reconfigurable SoCs (3DRSoCs). These platforms are composed of two layers that are vertically connected: the MPSoC layer and the FPGA layer. An example of a 3DRSoC is given in Fig 1.1. Stacking these two layers allows to conserve the characteristics of planar RSoC while inheriting the benefits of 3D technology. By using the vertical connections, the communication between the software code running on the MPSoC layer and the hardware accelerators running on the FPGA layer can be faster and better ensured. Therefore, 3DRSoCs seems to be promising solutions that better addresses the largest varieties of applications.

1.2 Problematic, Motivations and Objectives

The treatment of an application is often divided into tasks with dependencies between them (for example in Fig 1.1). Due to the complexity of applications, each task can have different software and/or hardware implementations. These implementations provide the opportunity for the tasks to be executed on the various components of the architecture. The software implementation of a task (or software task) is a piece of code executable on a processor. The hardware implementation of a task (or hardware task) is a synthesized and configurable function in the FPGA. From these task implementations, one of the challenge consists in managing the execution of all of them on the execution resources of the platform. One possible solution is then to embed an Operating System (OS) on the platform in order to organize the treatments of the application. The objective of an OS is to support several services, like the communication service, memory management, etc. and also the scheduling methods supporting an efficient management and use of computing resources. During the execution of an application on a 3DRSoC, the scheduling methods should be able to determine what resources (processor and/or FPGA area) are used by what task (spatial dimension) at what time (temporal dimension) in order to meet the communication cost constraints, the energy or power consumption budget, the overall execution time, etc. This problem, called spatio-temporal scheduling, is then more complex than the one of task scheduling for processor system.

For static applications, i.e. applications with a static execution flow of tasks, the spatio-temporal decisions can be taken offline, i.e. before the application starts running on the platform. In this case, we can ensure the optimality of these decisions. However,

for dynamic applications whose the behavior depends on external events, the spatio-temporal decisions for the tasks should be taken "online", i.e. during execution. In this case, the execution flow of tasks and the execution support for each task are not known a priori. Because of this "online" characteristic, we can not guarantee that these decisions will give the optimal solution. Therefore, online spatio-temporal scheduling strategies are proposed to find a "close to the optimum" solution.

The objective of my work consists in defining and evaluating a set of online spatio-temporal methods/strategies for 3DRSoCs. For this work, we propose to address several criteria, and to try to optimize them. The criteria and optimization are the following

- **Minimizing the global communication cost of the application**: in order to reduce the global communication cost of the application, the communication cost between tasks must be minimized. Although there exist different spatio-temporal scheduling algorithms for planar RSoC, taking into account the 3rd dimension of 3DRSoC makes the scheduling problem more difficult to solve and this problem is further complex when we consider the communication between tasks. As the tasks can be executed in software and/or hardware, they can communicate horizontally within a layer and/or vertically from a layer to another layer. The communications between tasks are linked to the time transfer between them and therefore related to the number of exchanged data during the communications. The more the communication distance between tasks is long, the more the communication cost will be penalized. Moreover, the interconnection of two tasks significantly affects the overall execution time of the application, the average load of the communication network and the power and energy consumed. For this reason, it is very important to propose scheduling strategies that take into consideration the placement of tasks on the three dimensions in order to minimize the communication cost between tasks, thus minimize the global communication cost of the application.

- **Minimizing the overall execution of the application running**: because the resources of the FPGA in a 3DRSoC are limited, it cannot accommodate all the tasks of the application at the same time. In this context, the MPSoC layer in 3DRSoC is used as a software support which offers the possibility of performing certain tasks as software. Thus, during the execution of the application, the execution of tasks can be anticipated on processors. Contrary to a software task whose execution

is done in a processor, the allocations and deallocations of hardware tasks at run-time can cause the FPGA fragmentation. This can lead to undesirable situations where future tasks can not be placed on the FPGA due to the bad placements of previous tasks, even there would be enough space. These tasks must be delayed until there will be available regions on the FPGA to accommodate them. Therefore, the overall execution time of the application will be increased and the overall performance of the system will be penalized. Instead of waiting for the task to be performed on the FPGA, the task could be executed on an available processor to anticipate their treatment and thus be completed sooner in time. In this case, a spatio-temporal strategy is necessary to support anticipation decision of software tasks and to be able to confirm or not the software anticipation when needed in order to minimize the overall execution time of the application.

1.3 Contributions

In 3DRSoC architectures as the one defined in Fig 1.1, the existence of the FPGA layer is crucial to accelerate the task processing while maintaining an easy and fast communication with the components of the MPSoC layer. Compare with software tasks, the management of hardware tasks on the FPGA layer is more complex and should be taken more carefully into account. On one hand, because the interconnection between hardware tasks consumes logical elements and routing signals. On the other hand, because a bad placement of a hardware task can prevent the placement of future tasks, thus penalize the overall execution time of the application. Making the use of FPGA more efficient is extremely important to achieve the best performance of the global 3DRSoC system. Therefore, before tackling spatio-temporal scheduling strategies for the 3D architectures, it is very important to study the spatio-temporal scheduling strategies for the FPGA layer.

In this work, two types of 3DRSoC architectures are considered: the 3D Homogeneous RSoC and the 3D Heterogeneous RSoC. The difference of these two 3DRSoCs comes from the FPGA layer architectures. In the 3D Homogeneous RSoC, the FPGA is a homogeneous reconfigurable resources model while in the 3D Heterogeneous RSoC, the FPGA is a heterogeneous reconfigurable resources model. The details of these two 3DRSoCs will be given later in this thesis.

To address the problematic of spatio-temporal task scheduling on the 3D Homogeneous RSoC and the 3D Heterogeneous, we organize our work into four steps. The first two steps consist in analyzing and proposing spatio-temporal scheduling strategies for the FPGA layer of these two 3DRSoCs:

- Step 1: we propose spatio-temporal scheduling strategies for hardware tasks executed in the homogeneous reconfigurable resources. These strategies aim at reducing the communication cost between tasks so that the global communication cost is minimized.
- Step 2: we address the heterogeneity of the reconfigurable resources. This heterogeneity imposes a stricter placement for hardware tasks, thus requires a different spatio-temporal strategy. We propose a strategy supporting this heterogeneity which aims at minimizing the overall execution time of the application.

Then, the step 1 and step 2 are served for addressing our main contributions which are in the step 3 and the step 4:

- Step 3: we extend the strategies proposed for homogeneous reconfigurable resources in step 1 to take into account the 3rd dimension of the 3D Homogeneous RSoC. Our strategy considers the 3rd dimension during the task scheduling in order to minimize the global communication cost of the application.
- Step 4: we address the spatio-temporal scheduling strategy for the 3D Heterogeneous RSoC. Our strategy exploits our previous work developed for the heterogeneous reconfigurable resources in step 2 and proposes to anticipate the software execution of a task if needed.

For all these proposed strategies, we have developed simulation models. A simulation tool has been developed and enables us to produce results. An hardware implementation of a complete platform is not part of this work but it is currently in progress, and will enable us to include our strategies in the scheduling service of an Operating System.

1.4 Thesis Organization

According to the objectives of our work and the different steps to address the global problematic, we organize this thesis in following chapters:

- Chapter 2 presents a background on the real-time system and gives the state-of-the-art overview of scheduling methods for MPSoC system, for different types of FPGA. The 3D technology and some scheduling methods on 3D platforms are presented as well.

- Chapter 3, addressing the step 1 of the contribution part, presents two online spatio-temporal scheduling strategies: the Pfair Extension for Reconfigurable Resource **(Pfair-ERR)** algorithm dealing with reducing the communication cost between tasks in a 2D bloc area FPGA and the Vertex List Structure Best Communication Fit **(VLS-BCF)** also aiming at reducing the communication between tasks but in a 2D free area FPGA. The results show that by limiting long and costly communications between tasks, the global communication cost of the application is significantly reduced.

- Chapter 4, addressing the step 2 of the contribution part, presents the online Spatio-Temporal Scheduling strategy for Heterogeneous FPGA **(STSH)** which deals with minimizing the overall execution time of an application running on this platform. STSH integrates prefetching technique while considering the priority of tasks and the placement decision to avoid conflicts between tasks. The results show that STSH leads to a significant reduction of the overall execution time compared to some non-prefetching and other existing prefetching methods. It also leads to a better FPGA resource utilization compared to others.

- Chapter 5, addressing the step 3 and 4 of the contribution part, presents the main contributions of this thesis. In this chapter, two online spatio-temporal scheduling strategies are introduced: the 3D Spatio-Temporal Scheduling **(3DSTS)** for the 3D homogeneous RSoC and the 3D Hardware/Software with Software execution Prediction **(3DHSSP)** algorithm for the 3D Heterogeneous RSoC. 3DSTS consists in considering the 3rd dimension during the scheduling and placement of tasks in order to minimize the global communication cost. 3DHSSP dealing with reducing

the overall execution time of applications, exploits the presence of processors in the MPSoC layer in order to anticipate a SW execution of a task when needed.

- Chapter 6 concludes this thesis and gives some perspectives

Chapter 2

Background and Related Works

As previously mentioned, the objective of this work concerns the definition of run-time task scheduling and placement for 3DRSoCs. However, in order to tackle the 3D systems, it is also necessary to analyze the influence of this issue on 2D systems such as: multiprocessor and reconfiguration architecture systems.

This chapter presents the background and the state-of-the art of the task scheduling and placement problem. It is composed of four sections. Section 2.1 presents real-time systems and discusses some existing scheduling methods for multiprocessor architecture systems. Section 2.2 introduces the reconfigurable architectures and presents a survey of existing techniques for task scheduling and placement for reconfigurable architecture systems. Section 2.3 presents the 3D technologies and related works for the 3D system. Finally, we conclude this chapter with our proposed approaches.

2.1 Real-time Systems

Real-time systems can be classified as two different categories: hard real-time systems or soft real-time systems. In hard real-time systems, all temporal constraints must be strictly respected. Any missing deadline will lead to catastrophic consequences. Hard real-time systems are used in military applications, space missions or automotive systems. Some examples of hard real-time systems are: fly-by-wire controllers for airplanes, monitoring systems for nuclear reactors, car navigation, robotics, etc.

In soft real-time systems, some temporal mistakes can be tolerated. They will decrease the quality of service, but they will not affect the correctness of the system. Web services, video conferencing, cell phone call are examples of soft real-time systems.

For these two types of real-time systems, the task scheduling is an important issue and large number of studies have been published in the literature. When the system architecture becomes more and more complex by including large number of heterogeneous processing cores, solving the task scheduling issue is critical and needs new scheduling strategies. Furthermore, when the system is submitted to large environment events, runtime decisions are necessary to support the dynamism of the application.

2.1.1 Offline and Online Scheduling

The scheduling service plays a very important role of an operating system. For a simple core processor, which executes just one task at a time, the scheduling has to determine the execution time of the tasks and manages the execution resource. For multiprocessors architectures, the scheduling is more complex by also determining the allocation of tasks on different execution resources.

Real-time task scheduling determines the order in which various tasks are selected for an execution on a set of resources. The real-time aspect consists in ensuring that each task respects its deadline execution time. To ensure this constraint, two different scheduling approaches are available for real-time systems: offline and online scheduling. Offline scheduling is applicable for applications where the execution flow of task set is known a priori. Thus, tasks are executed in a fixed order and this order is determined offline, i.e. before the system gets started. Offline scheduling is usually performed to find the optimal solution of tasks.

Contrary to the offline scheduling, the execution flow is not known in advance for online scheduling. All scheduling decisions are made on the fly without any knowledge of future arriving tasks. Online scheduling selects tasks to execute by analysis of their priorities. It is more flexible than offline scheduling since it can be used for the cases where the sequence of tasks dynamically changes at run-time. In almost cases, online scheduling algorithms try to produce an "approximate" solution, but can not guarantee the optimal solution.

2.1.2 Static and Dynamic scheduling

Most scheduling algorithms are priority-based and consist in determining the task priorities in different ways. There are two priority-based algorithms: static-priority and dynamic-priority.

Static-priority means there is an unique priority associated to each task. This priority is determined before the system runs and it will stay unchanged during the system execution. Among static-priority algorithms, Rate Monotonic (RM) [9] is known as the optimal algorithm. RM assigns priority according to the period, thus a task with a shorter period has a higher priority and will be executed first.

In dynamic-priority, the priority of a task is not fixed and can be changed during the execution. Earliest deadline first (EDF) [10], Least Completion Time (LCT) and Least-Laxity First (LLF) [11] are some examples of dynamic priority scheduling algorithms.

In the context of embedded systems, the dynamism requirement leads the designer to embed an operating system which supports dynamic scheduling. Static scheduling is often not implemented due to the inefficient processor usage.

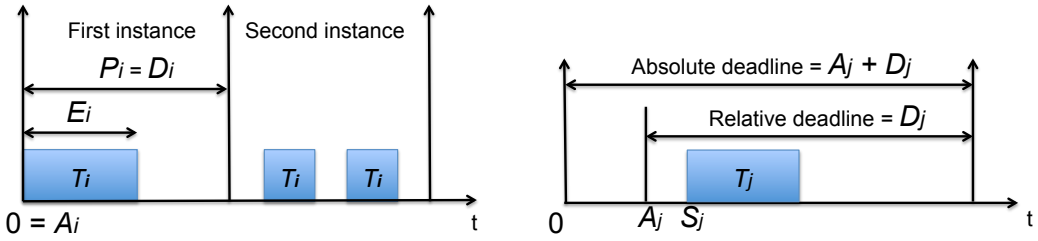
2.1.3 Type of tasks

Before presenting different types of tasks, we introduce here some basic task characteristics which would be useful for the understanding of this work.

- Task instance: each new execution of a task is called a task instance or a job. A task can be executed one or several times, i.e. a task can have one or several instances.
- Relative and absolute deadline: Absolute deadline is the time point at which the job should be completed. Relative deadline is the time length between the arrival time and the absolute deadline.

2.1.3.1 Periodic, Aperiodic and Sporadic tasks

Depending on the real-time application, tasks can be executed repetitively (in the case of reading the ambient temperature at regular intervals for example) or non-repetitive. A task can be classified into three following categories:



(a) periodic and preemptive task

(b) aperiodic and non-preemptive

FIGURE 2.1: -a-Example of a periodic and preemptive task; -b- Example of an aperiodic and non-preemptive task

- Periodic: a periodic task T_i is characterized by (A_i, E_i, P_i, D_i) with A_i as the release time (or arrival time), i.e. the time the task is ready to be scheduled, E_i as the worst case execution time (the maximum amount of time the task required to execute), P_i as the period of the task, D_i as a relative deadline ($D_i = P_i$ for almost cases). A job (or an instance) of the task is repeated indefinitely and the time length between two activations of successive instances is called "period". A job is released at the beginning of its period and must complete execution before the end of its period.

- Aperiodic: an aperiodic task T_i is characterized by (A_i, E_i, D_i) with A_i as the arrival time, E_i as the worst case execution time, and D_i as the absolute deadline. An aperiodic task must run at least once and it is not necessary to be repeated. In offline or static scheduling, the arrival time of a job is known before execution. In online scheduling and dynamic scheduling, the arrival time of a job is not known before execution, it will be computed on the fly. For an aperiodic task, the response time of a job is defined by the subtraction of the completion time and the arrival time of this job. In some cases, another parameter S_j, representing the time point that a task starts its execution, is also used to characterize the aperiodic task.

- Sporadic: sporadic tasks are a particular case of a periodic task. These are tasks repeated with a minimum period. A sporadic task T_i is characterized by: the arrival time A_i, the worst case execution time E_i, a relative deadline D_i and the minimum time MT_i between two successive jobs.

Fig 2.1(a) shows an example of a periodic and preemptive task T_i with T_i=(A_i,E_i,P_i,D_i). Another example of an aperiodic and non-preemptive task T_j with T_j=(A_j,E_j,D_j) is

shown in the Fig 2.1(b). The definition of a preemptive and non-preemptive task will be given just below.

2.1.3.2 Preemptive and Non-Preemptive tasks

Tasks are also distinguished by two types of execution: preemptive and non-preemptive execution. In order to respect the real-time constraints, it is generally necessary to use preemptive tasks, i.e. a task that can be interrupted by higher priority tasks and resumed to finish its execution later. However, preemptive tasks create overhead needed to switch between tasks. Non-preemptive tasks do not permit the preemption before the end of the job execution. This type of execution is easier to implement than preemptive execution one. Non-preemptive tasks guarantee exclusive access to shared resources and data which eliminates both the need for synchronization and its associated overhead [12]. For soft real-time applications, using non-preemptive tasks are usually more efficient than preemptive scheduling.

As previously mentioned, the context of embedded systems needs adaptative execution of tasks, and non-preemptive execution is generally not implemented due to the difficulty to rapidly react when a new event occurs.

2.1.3.3 Migrable and Non-Migrable tasks

Some advanced real-time applications require more than one processor to complete set of tasks efficiently and successfully. A migrable task is the term used when a suspended instance of a task may be resumed on different processors. Otherwise, every instance of non-migrable tasks always executes on the same processor.

The migration concept enables more flexibility for the execution, but this flexibility generates time overhead to move the context of task from one processor to another. When the processor cores are heterogeneous, migration of tasks is yet more complex because the task context must be transformed in order to be able to resume the task execution after the migration.

2.1.3.4 Dependent and Independent tasks

For almost applications, tasks are dependent and need to share data between them. Precedence constraints and data dependencies are often modeled by a directed acyclic graph (DAG). Formally, a DAG consists of several nodes (tasks) that can be executed on any available processors. An edge signifies that data produced by one task is used by another one. The number on an edge represents the cost of the communication, and can be the amount of exchanged data between two tasks. We call T_j the predecessor of T_i if T_i needs data from T_j to be executed. A task can have one or more predecessors and it is ready to execute when all of its predecessors complete their execution and produce their data. Fig 2.2 shows an example of a DAG comprising 5 tasks with the amount of exchanged data between tasks.

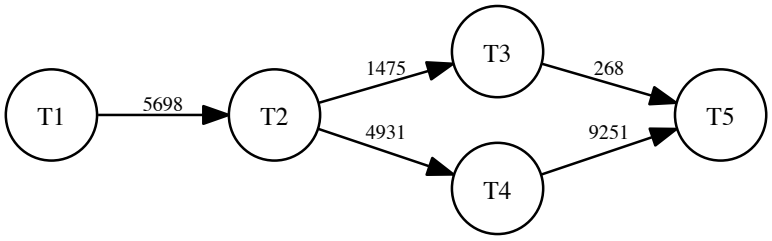


FIGURE 2.2: Example of a DAG task graph

2.1.4 Real-time Scheduling on Multiprocessors System

One of the main factors to measure the performance of multiprocessor systems is to analyze the scheduling of tasks in order to verify if the different processors are used as efficiently as possible at each time. Almost task scheduling algorithms consist in distributing a set of tasks among the processors to achieve the desired objective as: minimizing the schedule length, minimizing the communication cost between tasks or maximizing the processor utilization, etc. In this section, we introduce different multiprocessor systems and talking about the importance of scheduling in such systems. Existing scheduling algorithms are then presented and discussed.

2.1.4.1 Multiprocessors definition

A multiprocessor system is composed of many processors communicating with each other by an interconnect network. This network can be a communication bus or a network on chip (NoC). The simplest multiprocessors are described as in the Fig 2.3(a) where processors do not have any cache memory or local memory. Every read/write operation must use the communication bus to access to the shared memory. In such system, the bandwidth of the bus is limited as a processor must wait until the bus is idle in order to perform the read/write operation. A more evolved multiprocessor system has a cache added to each processor, this solution enables to reduce bus traffic toward the memory (Fig 2.3(b)). Another possibility is the system where each processor proceeds a cache memory and a local (private) memory (Fig 2.3(c)). In that case, every local variables, local data, constants, etc, are placed in the local memory. The shared memory is then only used for writable shared variables which will greatly reduce the contention for the bus. Fig 2.3(d) shows a multiprocessor interconnected by a NoC.

Considering the memory accesses, a multiprocessor system can be differentiated by two types: Uniform Memory Access (UMA) or Non Uniform Memory Access(NUMA). In UMA, each processor needs the same time to read a word from the shared memory. NUMA does not support this property, and the access time from/to the memory depends on the memory location relative to the processor.

2.1.4.2 Partitioning and Global Scheduling

Scheduling algorithms for multiprocessor systems are generally divided into two categories: partitioning and global scheduling. In partitioning scheduling, a task is only scheduled on a predefined processor and it cannot be moved to another processor during runtime. With partitioning method, the migration of tasks between different processors is not allowed. For all the cases, the scheduling scenario is well-studied before the system gets started and optimal partitioning of tasks is defined if it exists. This method reduces the complexity of the scheduler, but it can be inefficient in some cases where tasks must be migrated in order to respect several constraints of the system as: deadline, communication cost, temperature, energy, etc.

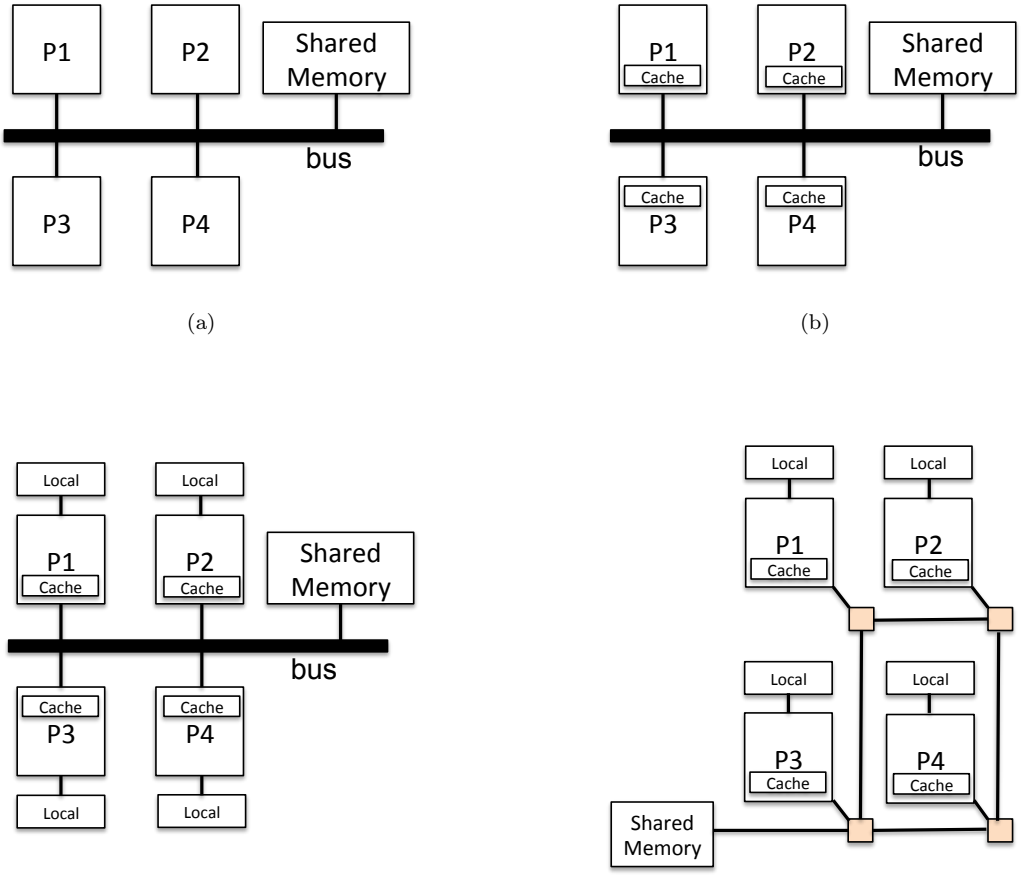


FIGURE 2.3: Multiprocessor systems with different data-communication infrastructures

In global scheduling, tasks are ordered by the scheduler during run-time. Tasks which are ready to be scheduled are stored in a single list (or queue) regarding the task priorities. Then, the highest priority task is selected by the scheduler to be executed on any available processor. Global scheduling allows task migration in order to satisfy the required constraints.

A semi-partitioned scheduling algorithm has been also proposed in [13], this algorithm limits the number of processors which can support a task execution. Thus, the penalties associated to task migration are reduced and the implementation of the system is less complex.

2.1.4.3 Performance parameters of scheduling algorithms

Many parameters can be used to evaluate the performances of scheduling algorithms. The basic parameters are following:

- Processor utilization: the performance of a scheduling algorithm is measured by the processor utilization, i.e. how all the processors are effectively used. The processor utilization is calculated by:

$$U = \sum_{i=1}^{n} E_i/P_i$$

  where E_i is the worst case execution time of T_i and P_i is the period of T_i (P_i is assumed to be equal to the relative deadlines D_i). A set of tasks is said schedulable in a uniprocessor system when and only when $U \leq 1$. It is said schedulable in a multiprocessor system of N_p processors when and only when $U \leq N_p$.

- Effectiveness: effectiveness of a scheduling algorithm can be measured by any factor for example: communication cost, overall execution time, energy, etc.

- Number of preemptions and migrations: it measures the number of preemptions of the tasks and the number of migrations that the tasks performed.

- Response time: the time duration between the time point when a task is ready to be executed and the time point when it finishes its execution.

- Deadline missed: the number of tasks which miss their deadline. This parameter is only used for soft real-time system, for hard real-time this case should not appear.

- Complexity: scheduling algorithm complexity is about how fast the algorithm performs. If a scheduling algorithm is complex, it may need a huge computation time and lead to delays in real-time systems.

2.1.4.4 Scheduling of independent tasks on multiprocessors

EDF [10] is known to be the optimal scheduling algorithm for independent and preemptive tasks in uniprocessor system. Fig 2.4 shows the EDF scheduling of 3 independent tasks T_1, T_2, T_3 in uniprocessor system. The characteristics of these tasks are described in the table 2.1. All the deadlines are met and the processor utilization is 100%.

Task	A_i	E_i	D_i
T_1	0	1	6
T_2	0	2	4
T_3	0	4	12

TABLE 2.1: Characteristics of three independent tasks T_1, T_2, T_3

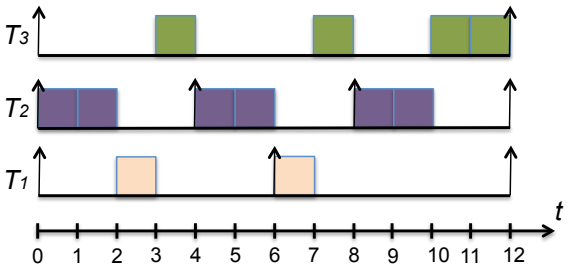


FIGURE 2.4: Earliest deadline first (EDF) scheduling in uniprocessor system

However, EDF is not optimal on multiprocessors since some deadlines may be not respected if EDF is used. Let's consider another example of scheduling 3 periodic tasks T_1=(0,2,4,4), T_2=(0,2,4,4) and T_3=(0,7,8,8) on two processors using EDF. T_1 and T_2 will execute on processor 1 and processor 2 because they have a higher priority than T_3. As a result, T_3 can only start executing from time 2 and will miss its deadline. But if T_1 and T_2 execute on the same processor (one after the other) and T_3 executes on another processor. Every task will meet their deadline.

Another dynamic scheduling algorithm, called Least-Laxity First (LLF) [11], is based on task laxity to define scheduling priority. The laxity of a task is defined by the subtraction of the deadline and the remaining execution time of this task. LLF is optimal for independent and preemptive tasks whose the relative deadlines are less or equal to their period. But LLF is still not optimal in a multiprocessor system.

Pfair [14] has been proved optimal for real-time scheduling of periodic tasks on a multiprocessor system. However, the migration and preemption costs are not taken into account in Pfair. In the worst case, a task may be migrated or preempted each time it is scheduled. For a real-time system, these can occur to significant delay and it can lead to degrade the performance of the system. To limit this problem, some other works based on Pfair are proposed to reduce at maximum the number of preemptions and migrations. Some of them are Generalized Deadline partitioned fair (DP-Fair) [15] or BFair [16].

A survey on real-time task scheduling is done in [17]. Fig 2.5 shows different scheduling algorithms with different parameters considered. However, almost of them consider that tasks are independent in the nature, i.e. no communication exists between tasks.

Scheduling Algorithm	Parameters considered
SPFPS	System utilization
E-Bfair	Scheduling points and scheduling overhead
ACO based metaheuristic	energy consumption
ZL policy	System utilization
SPHRTS	System utilization and schedulability
LADD	dynamic density
Allowance-fit	latest execution time
MSHS	CPU resources and schedulability
PDMS	system utilization
LRATA	Mean Time To Failure(MTTF)
EES	Energy consumption
E-Pfair	Deadline Miss Rate
Gang-EDF	Schedulability
TLS	Scheduling latency & CPU utilization
ReTAS	Jitter and scheduling overhead
SATA	synchronization penalties

FIGURE 2.5: Real-time task scheduling algorithms

2.1.4.5 Scheduling dependent tasks on multiprocessors

Scheduling of dependent tasks onto a set of homogeneous processors in order to minimize some performance measures has been quite studied for a long time. One of important performance measure is communication delays. Reducing the communication delays leads to not only the overall schedule length reduction but also the energy consumption reduction. If two communicating tasks T_i and T_j are executed on different processors, they will incur a communication cost penalty depending on the distance between two processors and the amount of exchanged data between two tasks. Otherwise, if T_i and T_j are executed on the same processor, the communication cost can be estimated to value zero. Smit et al [18] present a mapping algorithm to map an application task-graph at run-time. Communicating tasks are placed as closed as possible in order to save the energy consumption. Similarly, Singh et al [19] propose a number of communication-aware run-time mapping heuristics for the efficient mapping of multiple applications onto an 8 x 8 NoC-based MPSoC platform. Their technique tries to map the communicating tasks on the same processing resource and also the tasks of an application close to each other in order to reduce the communication overhead. Their proposed heuristics lead to reduction in the total execution time, energy consumption, average channel load and latency.

Two others important performance measures are the total duration of the schedule and the running time execution of the algorithm. There is usually a trade-off between these two performance measures: a proposed algorithm searching for the optimal solution incurs normally a high time running. Using Branch-and-Bound technique or integer linear programming (ILP) solver can find the optimal solution, but they need a huge computation time. Thus, many heuristics are proposed to quickly find an close-to-optimal solution in a polynomial-time. Some of them are based on list scheduling technique. The idea is to assign priorities to tasks and place the tasks in descending order of priorities in a list. The highest priority task is executed first. If some tasks have the same priority, several methods are used to decide in which order the tasks are executed. Some given priority functions are EDF, dynamic critical path, as late as possible, etc. Some of them based their algorithms on Tabu Search [20], Simulated Annealing [21] or ant colony optimization [22] technique in order to optimize the execution time.

Most of previous algorithms target offline scheduling. In online scheduling, if the entire execution flow is not known in advance, the scheduling decision must be done on the fly by using dynamic-priority of tasks. In this context, previous methods as ILP solver, Branch-and-Bound or other mentioned algorithms cannot be applied.

2.2 Task Scheduling And Placement for Reconfigurable Architecture

2.2.1 Reconfigurable Architecture

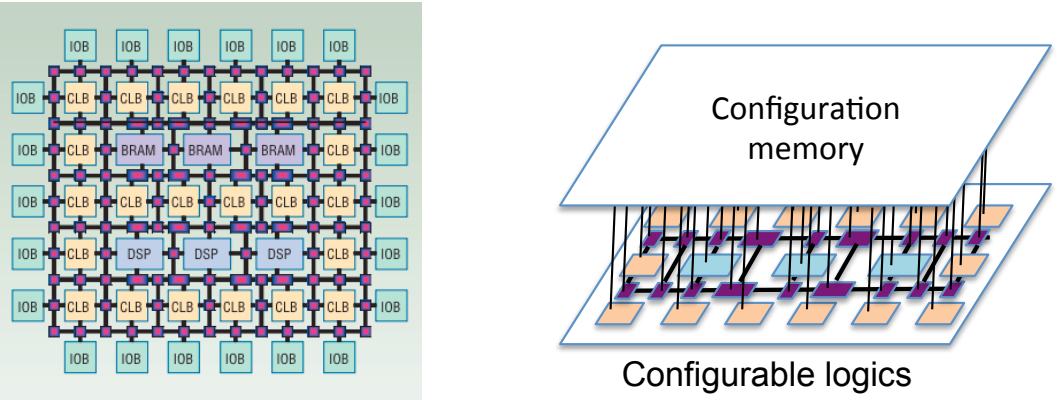
A configurable logic circuit is a circuit whose logic function can be configured. This type of circuit is made from basic logic elements and a set of them can be used to ensure a specific function. Reconfiguration concept means that it is possible to change the configuration if needed.

A circuit is said dynamically reconfigurable when it is possible to change its configuration during the execution of an application on the system. A circuit is said to be dynamically and partially reconfigurable when it is possible to change during the operation of the system and when the configuration can address a portion of the circuit, without interfering other running parts.

2.2.1.1 FPGA circuits

Several types of architectures exist and allow reconfiguration. The reconfiguration can be applied for operators of a processor or for hardware accelerators of a System-on-Chip (SoC) or in FPGA. Among them, FPGA is known as a very wised used reconfigurable architecture allowing a high configuration flexibility.

FPGA is a configurable logic device consisting of an array of configurable logic blocks (CLBs), programmable interconnects and programmable input/output blocks (IOBs). The CLB block is composed of Lookup Tables (LUTs), multiplexers (MUXs), Flip-Flops and registers. CLBs are connected together as well as to the programmable IOBs through programmable interconnects. Currently, FPGA can contain also Digital Signal Processing (DSP) blocks which have high-level functionality embedded into the silicon, and embedded memories (BRAMs). All these blocks are configurable to perform the required functionalities. Fig 2.6(a) represents a FPGA internal structure based on the Xilinx architecture style [23]



(a) (b)

FIGURE 2.6: -a-Example of a Xilinx architecture style, -b-Two layers representation of a reconfigurable architecture

FPGA can be represented as a circuit having two layers (Fig 2.6(b)): one layer for the architecture whose functionalities are configurable and another layer for the configuration memory. This memory stores all the configuration information and the FPGA is configured by charging the *bitstream* in this memory. A bitstream is a sequence of bits that are stored in the external memory of the FPGA as Compact Flash or internal memory

as BRAM. The configuration operation consists in the transmission of bitstream to the configuration memory via a reconfiguration controller.

2.2.1.2 Reconfigurable processor

Having a processor core into a reconfigurable circuit can improve the flexibility of the system, and can also offer the possibility to support software execution of an application. In this context, Xilinx FPGA devices introduce two types of processors: soft-core and hard-core. Soft-core processors, such as MicroBlaze or PicoBlaze, are implemented by using reconfigurable resources. Depending on the FPGA size, it can be possible to implement not only one soft-core processor but several soft-core processors. Hard-core processors, such as PowerPC or ARM, are hard-wired on the FPGA die. Depending on the FPGA devices, the number of hard-core processors is fixed: no hard-core processor, 1, 2, etc. The objective of a processor is to control the reconfiguration process, the communication with peripherals and the memory management. Also, processors can be in charge of executing software code if needed.

2.2.1.3 Dynamic and Partial Reconfiguration

One of the main FPGA research interest is dynamic and partial reconfiguration. Different levels of reconfiguration can be defined: the complete reconfiguration supports the reconfiguration of the full FPGA logic resources to perform a design function ; dynamic and partial reconfiguration allows a system to change partially the FPGA logic resources on the fly while the remaining logic resources continue to operate without interruption. Partial reconfiguration is not supported on all FPGAs, one of the biggest companies offering the partial reconfiguration capability is Xilinx with their Virtex families.

Depending on the technology supported by Xilinx, partial reconfiguration can be done by module-based style or difference-based style. Module-based is able to reconfigure distinct modular parts of the design. Each module is determined by a separate partial bitstream. A module is also called a "task". A partial bitstream contains configuration information for a specific area on the FPGA. By sending the partial bitstream to Internal Configuration Acces Port (ICAP), the configuration memory of the FPGA is partially changed and the FPGA is partially reconfigured to perform the task function. In difference-based

style, the FGPA is partially changed by sending a partial bitstream that contains only information about differences between the current and the new content of the FPGA. However, in this thesis, we focus only on module-based style of partial reconfiguration.

In module-based reconfiguration, the FPGA logic resources are partitioned into static part and reconfigurable part. Static part is configured during the circuit initialization and it stays unchanged during all the FPGA utilization. It can be operated and contains logic resources controlling the reconfiguration process as well as the communication with the reconfigurable part. Reconfigurable part contains logic resources that are reconfigured independently of the static part. Reconfiguration concept can be based on the three following techniques:

- Reconfiguration by column: The reconfigurable part is composed of columns which have all the same size. These columns can be reconfigured separately from each other, but they must be reconfigured entirely. Hardware tasks are synthesized by design tools to occupy one or more columns. In this model, if a task does not occupy the whole area of a column, the remaining free area of the column is wasted. Fig 2.7 shows an example of T_1 executing on the column C_1 and T_2 on the column C_2. The columns C_3, C_4 are free to receive other tasks. This model is used by the first FPGA Xilinx as: Virtex, Virtex-II and Virtex-II Pro.

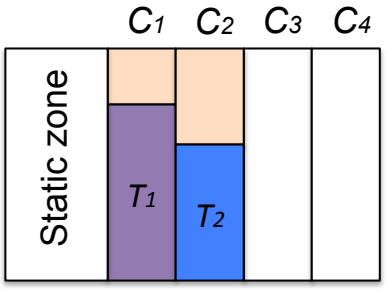


FIGURE 2.7: Reconfiguration by column. T_1 is executed on C_1 and T_2 on C_2

- Reconfiguration by slot: similar to the reconfiguration by column case, this model is composed of same size slots as in Fig 2.8. The remaining area of a slot is wasted if the area occupied by the task is smaller than a slot.

- Reconfiguration by region: Instead of reconfiguring by column which creates wasted area for a task, Xilinx has evaluated the reconfiguration technology by giving the possibility to reconfigure the circuit by regions. With this technique, the user has

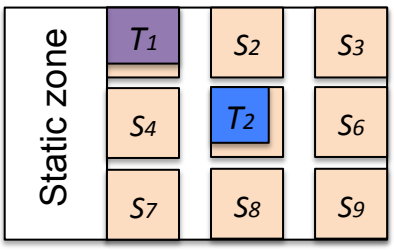
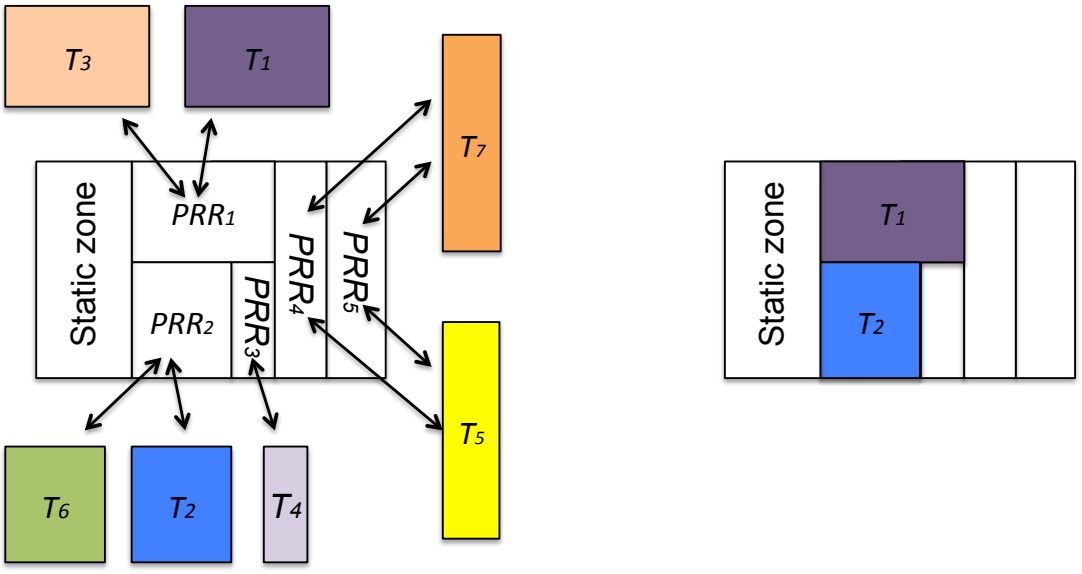


FIGURE 2.8: Reconfiguration by slot. T_1 is executed on S_1 and T_2 on S_5

the possibility to split the reconfigurable resources in different size regions at the circuit initialization. Hardware tasks are synthesized by design tools to be executable on one or several predefined regions. These regions are called Partially Reconfigurable Regions (PRR). All tasks for a given PRR must respect connection constraints relative to that PRR. Fig 2.9(a) shows an example of five $PRRs$ and tasks associated to each PRR. Fig 2.9(b) shows a case where T_1 is configured on PRR_1 and T_2 is configured on PRR_2 during runtime.



(a) (b)

FIGURE 2.9: Reconfiguration by region -a- Placement possibilities of relocatable tasks on different PRR, -b- Example during runtime, T_1 executes on PRR_1 and T_2 on PRR_2

Fig 2.10 shows a typical structure of a microprocessor-based system where the reconfiguration process is controlled by a microprocessor (Microblaze or PowerPC). All partial bitstreams of tasks are stored in the annex memory which can be external memory as

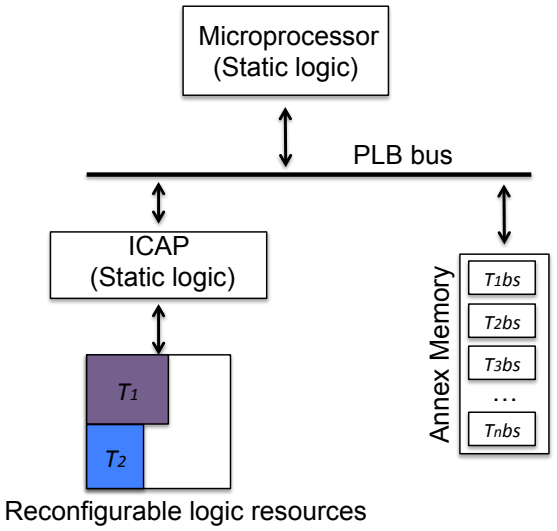


FIGURE 2.10: Microprocessor-based system controlling reconfigurable resources

Compact Flash or internal memory as BRAM. The microprocessor takes in charge of reading of partial bitstreams from the annex memory and then sending them to the ICAP. A bitstream of a task contains the information about regions that the task can be executed on. Once the bitstream is sent to ICAP, the corresponding region is reconfigured to perform the task function. The microprocessor, ICAP and PLB bus containing static logic resources are located in the static zone of the FPGA. Any connection between tasks and static logic is passed through a *MacroBus*.

2.2.1.4 Hardware Task Characteristics

If a software task is a piece of code or data which is executable on a processor, a hardware task is a synthesized logical function executable on the FPGA. Therefore, apart from similar parameters as a software task, a hardware task need to be defined by other parameters as: its occupied area, its reconfiguration time and its worst case execution time. The occupied area presents the number of resources needed (logic resources and routing resources) to implement the task functionality in hardware. The reconfiguration time presents the time needed to load the bitstream of the task to the configuration memory through the ICAP. The reconfiguration time can be calculated precisely by the size of the bitstream (number of bits) and the throughput of the ICAP (number of bits/second). The worst case execution time represents the time needed to process the task functionality on the FPGA.

2.2.1.5 Hardware Task Relocation

The idea of task relocation, called Partial Bitstream Relocation (PBR), is to be able to move a task from a region to other identical regions with only one pre-compiled partial bitstream of the task. Standard DPR does not support this kind of task relocation. In previous DPR design flows described by Xilinx, a bitstream of a task can only be generated for one predefined PRR. A hardware task is executable on another PRR when and only when another bitstream of the task has been generated for that PRR. Different bitstreams realizing the same task function must be stored in the external memory and the choice of bitstream to used is decided during the execution of the application. Disadvantages from this standard DPR are: i) the storage capability of the external memory needs to be huge to store all generated bitstreams ii) it requires big efforts from the application designer, i.e. applications needs to be carefully planned before execution, resulting in systems with a limited flexibility.

Several works have addressed this issue in order to try to solve the PBR problem. One of the first significant tool allowing to generate a partial bitstream from the complete bitstream of a reconfigurable by column FPGA is PARBIT [24]. The tool enables the bitstream to be relocated to a different predefined column of the FPGA. However, the generation of bitstream is done off-line which is not suitable for an on-line context. Another tool supporting the relocation by column on a Virtex-E is REPLICA [25]. Different to PARBIT, REPLICA works as a hardware relocation filter which transforms the bitstream when it is loaded from the external memory. REPLICA allows to execute the precompiled bitstream on a desired column during the application execution, i.e. online. REPLICA2Pro [26] is similar to REPLICA but it targets Virtex II and Virtex II pro devices. Another hardware-base relocation filter is BiRF [27] which targets not only Virtex II Pro but also Virtex 4 and 5.

Authors in [28] propose an Off-line/On-line Relocation of Bitstreams (OORBIT) based on a combination of off-line and on-line PBR approaches. By using their development tool, they analyzed first the bitstream and find off-line all the possible relocatable areas in the design. Secondly, for each possible area, the Frame Addresses Register (FAR) and the Cyclic Redundancy Check Register (CRC) are calculated. Then, during the application execution, the FAP and CRC addresses of the original bitstream are modified by the values calculated offline which correspond to the new desired allocation.

2.2.2 Online scheduling and placement of hardware tasks for Reconfigurable Resources

Due to the limited resources of the FPGA, all tasks can not be accommodated simultaneously. It is then necessary to schedule tasks over time on the FPGA. If a task can be allocated to any available processor in an homogeneous multi-processor system, a hardware task can only be allocated to the FPGA when a feasible region containing sufficient contiguous area is available to accommodate the task.

Similar to software tasks, the management, scheduling and placement of hardware tasks in the FPGA, also need to be considered. These services are managed by the free resource manager, scheduler and placer in an Operating System(OS) for Reconfigurable Systems. Apart from these services, OS provides also others services such as: loader, inter-task communication and synchronization, memory management, and interface to easy the task development. This thesis focuses on the task scheduling problem, and considers temporal as well as spatial problem.

Scheduler : Due to the limited resources on the FPGA, all the tasks cannot use these resources simultaneously. Scheduler decides which task must be executed and when. The scheduling decision is based on specific scheduling policy (offline or online scheduling, static or dynamic priority) and on task characteristics (preemptive, periodic, etc.)

Placer: once the scheduler decides to execute which task and analyses information about this new task, the placer is responsible to choose the feasible position for the task according to some objective function. If a feasible position exists, it requests the Loader to load the task bitstream through ICAP. At the same time, it also requests the free space manager to update the free resources.

Free Space Manager : The free space manager is responsible to maintain a data structure to keep track of unoccupied area. It must communicate with the Placer in order to update the free resources when a task is added or removed from the FPGA.

2.2.2.1 Task scheduling for Reconfigurable Resources

Beside adopting conventional scheduling algorithms as multiprocessor system, some authors proposed other scheduling methods for Reconfigurable Resources. Danne et Platzner

[29] propose EDF-NextFit and Merge-Server Distribute Load (MSDL) algorithms scheduling for preemptive and periodic tasks on the FPGA . In EDF-NextFit (EDF-NF), task with the earliest deadline is scheduled first. However, if there is no available region to accommodate the task for the requested time, EDF-NF continues to schedule next tasks. This method improves hardware resource utilization but it is not practical for a large amount of tasks. Thus, the authors propose the MSDL which uses the concept of servers to target a large set of tasks. MSDL reserves area and execution time for other tasks. Tasks are successively merged into servers which are then scheduled sequentially. But in terms of hardware resource utilization, MSDL is not as efficient as EDF-NF.

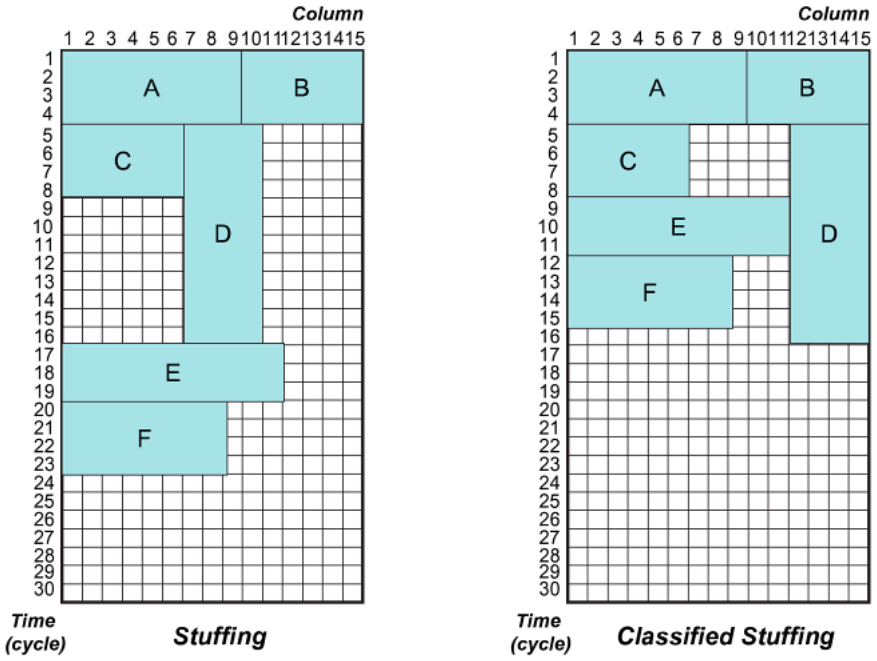
Several scheduling algorithms based on configuration prefetching technique tries to reduce the reconfiguration overhead for dependent hardware tasks, thus minimizes the overall schedule length of the application. The principle of prefetching is to load a hardware task as soon as possible whenever the reconfiguration port is available and a feasible position exists to accommodate the task. Even if the task cannot be executed immediately after its reconfiguration due to involvement of dependencies with other tasks, the fact of hiding the reconfiguration phase by loading the task during the execution of other tasks reduces significantly the reconfiguration overhead. Redaelli et al [30] propose off-line Napoleon technique to schedule dependent tasks of a direct acyclic graph (DAG) on the FPGA. Napoleon considers three methods: task reuse, prefetching and anti-fragmentation placement. Task reuse means two hardware tasks of the same type have the possibility to be executed exactly on the same region on the FPGA, with a single configuration at the beginning. For the anti-fragmentation placement, a task is placed to the furthest feasible position from the FPGA center. They experimented also with a system implementing two reconfiguration ports ICAP, thus until two tasks can be reconfigured at the same time. Resano et al [31] present a hybrid prefetch heuristic scheduling for runtime reconfigurations, but carring out the scheduling computations at design-time. Qu et al [32] present three static based schedulers using prefetching, the first is developed from list-based schedulers where each task has a priority representing the urgency of the configuration. The second is based on constraint programming and the last uses a guide random search strategy.

Very few scheduling algorithms address communication cost between tasks as a specific constraint, even if it has great influence on the latency and network loads. Lu et al [33] present a CASA algorithm which considers data communication between hardware tasks

and external devices. Mahr and Bobda [34] introduce an online scheduling algorithm to reduce communication cost (between tasks) on dynamic Networks-on-Chip through runtime relocation of tasks. In their method, a task can be preempted, relocated and placed to the region for which the communication costs are minimized. Gohringer et al [35] present a priority scheduling algorithm in which the scheduling decisions are based on task communications but the proposal is a static strategy.

2.2.2.2 Task Placement For 1D Area Model

Steiger et al [36][37] introduce the horizon and stuffing techniques for 1D model and presented also the extension of these techniques to work on the 2D model. The "horizon" makes sure that arriving tasks are only scheduled when they do not overlap in time or space with other scheduled tasks. The "stuffing" always places an arriving task on the leftmost column (in the case of 1D model) or leftmost area of its free space (in the case of 2D model). As a consequence, it may create the fragmentation in FPGA.



(a) (b)

FIGURE 2.11: Stuffing and Classified Stuffing. The figure is taken from [1]

Chen et al [1] propose a task placement method called classified stuffing technique to reduce the fragmentation on a 1D structure. Thus, the arriving tasks will be classified into two types: i) low space utilization ratio (low SUR), in this case the execution time

is long but the required columns are less ; ii) high SUR, in this case the execution time is short but the required columns are important. Depending on the arriving task type, it will be started from the leftmost (if the task is a low SUR) or rightmost column of the FPGA (if a task is a high SUR). The classified stuffing technique gives a smaller fragmentation and allows a shorter overall schedule time. Fig 2.11 shows an example of Stuffing method and Classified Stuffing. Task A, B, C, E, F are high SUR and task D is low SUR. For Horizon and Stuffing technique, task D is placed horizontally on the left next to task C, thus task E and task F cannot be placed due to the insufficient columns on the FPGA. However, the classified stuffing technique places the task D on the right most column which eases the placement of E and F.

Hubner et al [38] divide the structure of Virtex-II architecture into vertical configuration slots. Thus, a task can be placed in any slot. They also proved the possibility to place different tasks on top of each other if the sum of these tasks does not exceed the height of the slot. In this paper, the memory configuration and the communication manner are well detailed.

2.2.2.3 Task Placement For 2D Bloc Area Model

As 2D Bloc Area Model contains a statically-fixed number of slots and because each slot can accommodate at most one task at a time, there is no placement issue for this model. The most difficult comes from the scheduling of tasks on these slots.

2.2.2.4 Task Placement For 2D Free Area Model

The placement algorithms for hardware tasks onto 2D Free Area FPGA must deal with two aspects: Free space management and task placement.

Online Free Space Management: The literature shows a large number of methods and algorithms for the management of free space. One of the first authors to introduce on-line free space management algorithm for 2D Free Area Model is Barangan et al [39] with Keep All Maximum Empty Rectangle (KAMER). In this method, the FPGA is modeled as a big rectangle and arriving tasks are modeled as small rectangles which will be allocated inside the FPGA. Kamer holds a list of empty overlapping rectangles, arriving tasks will be allocated on these empty rectangles without overlapping with other

allocated tasks. Fig 2.12 shows how Kamer works. The bitstream of three tasks T_1, T_2 and T_3 are stored in the external memory, their required resources are presented on top of the figure. With the allocation of T_1 on the FPGA, Kamer creates two MERs (Maximum Empty Rectangle) noted R_1, R_2. Task T_2 can only be fitted in R_2, we suppose that T_2 will be placed at the Bottom Left of R_2 at its arriving. R_1 and R_2 will be split to create 3 new MERS note R_1', R_2' and R_3'.

The drawback of Kamer is its complexity which is $O(n^2)$, i.e the amount of MERs increases quadratically with the amount of currently executing tasks. Thus, they proposed an alternatively method to reduce the complexity to $O(n)$, this method is called Keep Non Overlapping Empty Rectang (Kner). Kner has two ways to create NERs: horizontal et vertical. Horizontal Kner tries to hold the width of empty rectangles as max as possible while vertical Kner holds the height of empty rectangles (see Fig 2.12). However, the cost for this improvement is, in some cases, an arriving task can not be placed even there would be enough space, for example in the case of horizontal Kner in Fig 2.12.

Many other methods are proposed to improve Kamer and Kner methods or identify the MERs et NERs in different ways with a low complexity. Handa et al [40] propose to construct MERs by using a staircase data structure method. Likewise, Cui et al [41] propose the Scan Line Algorithm (SLA) and Lu et al [42] propose Flow Scanning (FS) for finding MERs. Arguing that the amount of empty rectangles grows faster than the amount of occupied rectangles, Ahmadinia et al [43] propose to keep track of the occupied rectangles instead of MERs.

Few other methods do not maintain the list of free rectangles but a list of vertex or edges formed by current tasks to detect free areas, Skyline envelope [44] heuristic and VertexList [45] are two interesting examples. Skyline was initially a heuristic for bin-packing problems using horizon or "skyline" edges formed by the topmost edges of already packed rectangles to keep track of the free areas. Vertex List Structure (VLS) uses a list of vertices to describe the boundary between the free and the occupied FPGA. The advantages from these two methods compared to MERs are a simplified data structure and a simple list to manage. The complexity for VertexList is $O(n+4)$ when the amount of tasks on the FPGA is high. An example of VLS is given in Fig 2.12. When T_1 is executed on the FPGA, VLS is represented by a set of six vertices. The arriving task

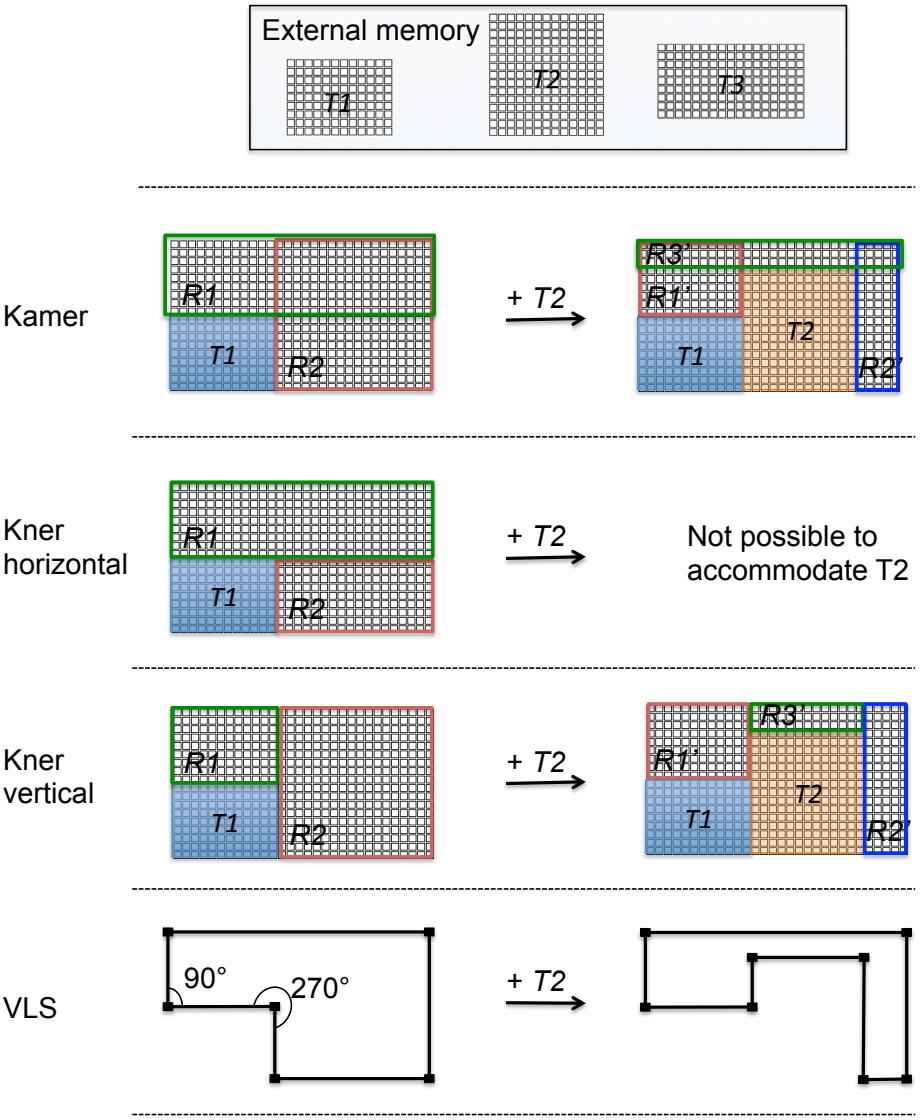


FIGURE 2.12: Different algorithms to manage free space of the FPGA at runtime

will be placed at one of these vertices. If T_2 is placed at the lowest left vertex, VLS will be modified and now contains eight vertices.

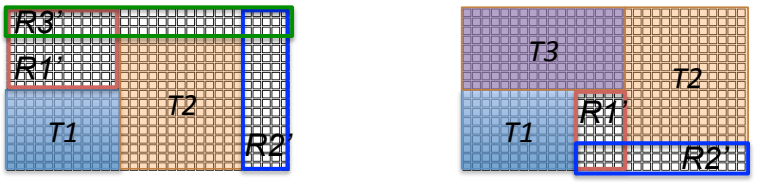
Olakkenghil and Baskaran [46] propose a new data structure based on a run-length encoding to manage empty areas. In their work, the FPGA surface is modeled by a matrix coded according to reflected binary gray curve.

Placement: Once the free space is found, the choice of placement must be decided, i.e where to place the task, on which corner of which available rectangle (in the case of Kamer) or on which vertex (in the case of VLS), etc. This placement decision is made according to the desired objective, for example: minimize the FPGA fragmentation, minimize the rejection ratio of tasks, increase the FPGA utilization, minimize the

communication cost, etc.

Vertices in VLS can be distinguished by two types: 90 degree-angle vertex and 270 degree-angle vertex. However, only the first type (90 degrees) is considered for the task placement due to the lower value of fragmentation compared to another type. J.Tabero et al [45] use Vertex List with BestFit-Fragmentation heuristic to estimate the fragmentation parameter. The arriving task will be placed on the 90 degree-angle vertex giving the lowest fragmentation value. By ignoring this parameter, sufficient contiguous area may not be available for placement of arriving task in the near future, even if total empty area may be greater than the task area.

For methods based on Kamer, some authors used First Fit, Best Fit, Worst Fit as the ways to fit new arriving tasks according to a given criterion. First Fit is used to reduce the searching time by placing a new task on the first found MER. Best Fit finds the smallest MER while Worst Fit finds the largest MER that the new task can be fitted. These methods are usually combined with Bottom Left, Bottom Right, Top Left, Top Right methods to choose the MER corner for placement of a new task. Fig 2.13 shows a case where the Top Right method gives a lower FPGA fragmentation than the Bottom Left methods. Therefore, T_3 can be allocated to the FPGA in the Fig 2.13(b)



(a) (b)

FIGURE 2.13: -a- The placement of T_2 creates the FPGA fragmentation and prevents T_3 not to be placed -b- The placement of T_2 creates a low FPGA fragmentation and allows T_3 to be placed

Marconi et al [4] propose Quad-Corner (QC) method for task placement in order to increase the FPGA utilization and system performance. An arriving task is placed close to the four corners of the FPGA. With this method, a large free area will be ensured in the middle of the FPGA for allocating future tasks.

The data structure used by Olakkenghil and Baskaran [46] allows a fast update when a task is inserted or deleted from the FPGA. They propose a new fragmentation metric giving an indication of continuity of free space. The arriving task will be placed by using

first fit mode or fragmentation aware best fit mode in order to minimize the task rejection ratio and increasing FPGA utilization.

Another placement strategy, called Nearest Possible Position (NPP), is proposed by Ahmadinia et al [43] to reduce the communication cost. Therefore, a task will be placed as close as possible of its communicating tasks without overlapping with currently executing tasks. Mahr and Bobda [34] use the same idea of NPP, they propose an algorithm which allows currently running tasks on the FPGA to be preempted, resorted and replaced to minimize the communication cost. However, they did not consider the preemption cost which is non negligible.

2.2.2.5 Task Placement For 2D Heterogeneous Model

For our knowledge, most of hardware task placement algorithms deal with 1D or 2D FPGA homogenous architecture (like 2D Bloc Area FPGA and 2D Free Area FPGA). However, real FPGAs contain not only CLB blocks but also embedded static components (as BRAM blocks, multipliers and DSPs) in a certain disposition and this heterogeneity imposes stricter placement constraints for the task. A task including static components can not be placed anywhere on the FPGA because their feasible positions are limited by the locations of static components on the FPGA. A feasible position for the task is the region on the FPGA whose all resources are available and matching those in the task. An example of feasible positions for T_1 which contains several CLBs and one static block is shown in Fig 2.14(a). Fig 2.14(b) shows feasible positions for T_2 which contains only CLBs.

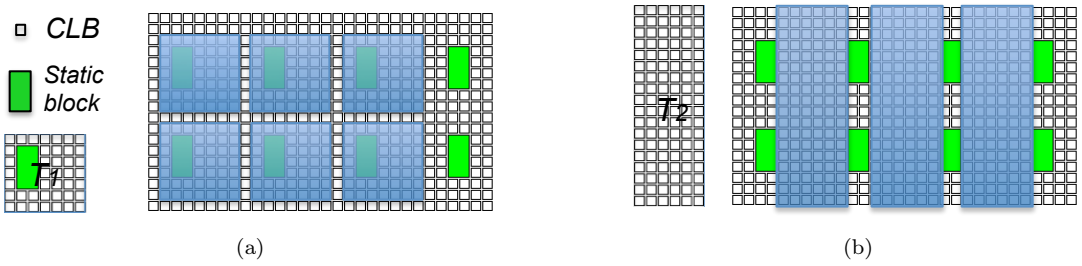


FIGURE 2.14: -a-Feasible positions for T_1, -b- feasible positions for T_2

Few algorithms deal with task placement on 2D heterogeneous architecture. Koester et al [47] propose a placement algorithm which is able to deal with the constraints of the hardware tasks. For each hardware task, the given set of feasible positions is predefined

at design time but not at run-time. However, they did not mention the way to calculate these feasible positions. They presented two algorithms : i) the Static Utilization Probability Fit (SUP Fit) which generates the position weights for each task before run-time and therefore uses constant position weights at run-time ii) the Runtime Utilization Probability Fit (RUP Fit) which updates dynamically position weights at run-time according to the currently allocated hardware tasks. Based on the value of position weights, these two algorithms will choose the best feasible position to place new tasks.

Another placement method is presented by Eiche et al [48]. By using a discrete Hopfield neuronal network, an on-line placer for heterogeneous devices was implemented. They consider that the FPGA is divided in several Partial Reconfigurable Regions and a task must contain at least one of them. This consideration can create a waste of resources when a task doesn't need the entire resources in the PRR.

Huriaux et al [49] present a 2D heterogeneous FPGA architecture supporting the task migration. Their FPGA uses long line connections to allow tasks to "float" horizontally within a region, thus greatly expanding the feasible regions for a task. In their enhanced FPGA architecture, a task can be placed in a region whose the number of resources are sufficient but not necessary matching for the task

2.2.3 Spatio-Temporal Scheduling for Reconfigurable Resources

From all previous works, we can see that scheduling and placement must not be split and solved separately in order to exploit at maximum the resources available by the FPGA. Considering only the scheduling order of tasks without their placement can lead to FPGA fragmentation. Moreover, it may increase the chip temperature, the power consumption, the communication cost, etc. Then considering only task placement but not the task priority scheduling will lead to suboptimal solutions. Therefore, a task should be managed in time (temporal scheduling) and in space (spatial scheduling). In this context, we talk about spatio-temporal scheduling algorithm which is a combination of scheduling algorithm and placement algorithm.

A spatio-temporal scheduling algorithm must answer to all the questions of a scheduling algorithm and a placement algorithm, i.e.:

- what are the next tasks to schedule for the next time cycles ?

- what are the feasible regions for the tasks and what are the best feasible regions to place them ?

Depending on the targeted objective, the answers to these questions will be different. Some examples of spatio-temporal scheduling algorithms are: Horizon and Stuffing [36][37], or Classifed Stuffing [1] aiming at reducing the FPGA fragmentation, or Napoleon [30] aiming at reducing the overall execution time length of the application, etc. More spatio-temporal algorithms can be found in [50].

2.2.4 Hardware/Software Partitioning and Scheduling for Reconfigurable Architecture

A Hardware/SoftWare (HW/SW) co-design in reconfigurable system can be characterized by the presence of a set of m processors and one (or several) reconfigurable resources (FPGA for example). In a HW/SW system, executing a task in HW is usually faster but less flexible compared with a SW execution. To achieve high performance system, HW/SW partitioning and scheduling algorithms are necessary to decide which tasks should be implemented in HW or SW, at what time, on which processors or in which region of the RR (Reconfigurable Region).

Traditional offline ways to solve the HW/SW partitioning and scheduling problems are based on HW-oriented and SW-oriented algorithms. In HW-oriented (respectively SW-oriented) algorithm, for example the one proposed in [51] and [52], a complete HW (respectively SW) solution is generated first and then many iterations are run to move tasks into the SW (respectively HW) until the performance constraints are fulfilled. Liu et al [53] propose a hybrid algorithm derived from Tabu Search (TS) and Simulated Annealing for solving HW/SW partitioning and scheduling problem. They applied two concepts called early partial reconfiguration and incremental reconfiguration. Redaelli et al [30] propose an exact ILP formulation for the task scheduling problem on 2D homogenous reconfigurable FPGAs to minimize the overall latency of the application. This work is then extended for a HW/SW scenario.

Contrary to the offline scheduling, online HW/SW scheduling needs to decide the order of task execution at runtime. As a result, it may increase runtime overhead. Al-Wattar

et al [54] propose two heuristics for online scheduling of hard real-time tasks for partially reconfigurable devices and implemented them on a system with two processors and five Partial Reconfigurable Regions (PRR). Kalte et al [55] address the multitasking in reconfigurable systems and discussed ways to relocate HW tasks during runtime. Their method requires to save and restore the state information of a HW task during runtime. Shang et al [56] propose HW/SW scheduling algorithms to optimize multiple costs as execution time, system cost and average power consumption.

Teich presented in [57] major achievements on methods and tools for HW/SW codesign until today and predicted in which direction research in codesign might evolve in the future.

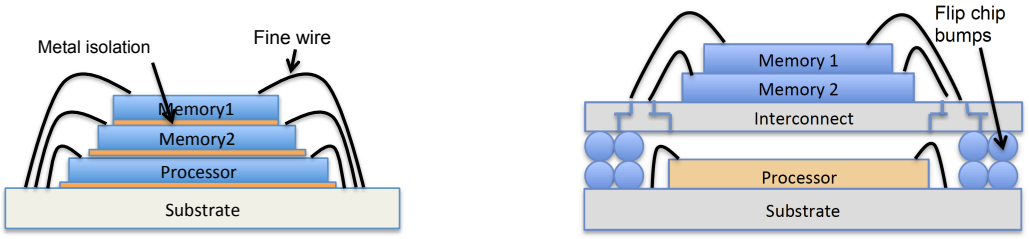
2.3 Three-dimensional architectures

2.3.1 3DSiP and 3DPoP

3D Silicon-in-Package (3DSiP) is known as the first used 3D technology. The idea of 3D SiP is to put in the same box different components for which the connections between components are done between their pads I/O via the wide of "bonding" [58][59], see Fig 2.15(a). Generally there is an extra layer on which no component is connected, it serves to create an interface between all layers and pins for connection to the outside. For 3DSiP, smallest components must be chosen in order to reduce the size of the 3DSiP. The advantage of 3DSiP is to use commercial components already encapsulated in their case. A 3DSiP stacked as above processor, a RAM module and a graphics processor is entirely possible.

Another 3D packaging innovation is Package-on-Package (3DPoP) that involves the stacking of two or more packages on top of one another. In 3DPoP, signals are routed between the packages through standard package interfaces. Obviously, one advantage of this vertical combination of the different packages is board space savings. Fig 2.15(b) is an example of 3DPoP technology which contains the processor package on the bottom, memory package on top.

These 3DSiPs or 3DPoPs are already commercially available products (for example Amkor in the U.S. and STATS ChipPAC in Singapore). These technologies are present



(a) (b)

FIGURE 2.15: -a- Example of a 3D Silicon-in-Package (3DSiP), -b- Example of a 3D Package-On-Package (3DPoP)

in various recent application processors in the mobile world to stack the DRAM on top of the processor. Today's new cell phones have at least one SiP or PoP. Apple A4, A7, A8 also used these technologies.

Despite its benefits, the challenges of this technology are in term of form factor and wire length. In addition, the small number of vertical connections are limited by the number of pads on the I/O ring and the inter-die communication speed. Therefore, 3D die stacking and 3D monolithic are found to resolve the challenges of SiPs or PoPs.

2.3.2 3D Integrated Circuit

Three dimensional integrated circuits (3D IC) has been considered to solve the challenges of SIP and PoP. It refers to a stack multiple layers of IC over each other though vertically bonded micro-bumps or through silicon vias (TSV) as shown in Fig 2.16. Each layer can be fabricated and optimized using their respective technologies and assembled to form a vertical stack. Depending on the applications, TSV's diameter may vary from 1um to 10um, their form factor (height to width ratio) from 1 to 30 and a minimum pitch of 10um. Smaller TSV sizes give the highest integration density. To be useful in multiprocessor communication (through NoC for example), TSVs should be used in a group or bundle, but not as a single communication point [60][61].

Another key differentiator in 3D IC integration is related to the stacking orientation. One option is Face-to-Face (F2F) alignment where stacked circuits are assembled so that the metal layers are face to face, see Fig 2.16(b). This option gives a high integration density, since there is no TSV in the active area. However, it is not possible to stack more

than two layers this way. Another option is Face-to-back (F2B). In this case, integration density is lower, but stacking more than two layers is possible. Not all 3D IC stacks need to use TSVs connection. If there is only two chips and if peripheral connections are sufficient for signal I/O, power and ground connections, then only Face-to-Face bonding is needed. We need to take in account the cost of 3D stacking and integration of TSVs because they consume silicon area. An example of F2F and F2B is given in Fig 2.16(b).

There is also another architecture called 3D monolithic where two MOS transistors, placed at different silicon layers, are connected directly one above the other. This concept is completely different, and for this technology, the thermal problem and the manufacturing cost are prohibitive compared to medium and high density 3D. We will not discuss about 3D monolithic in this work.

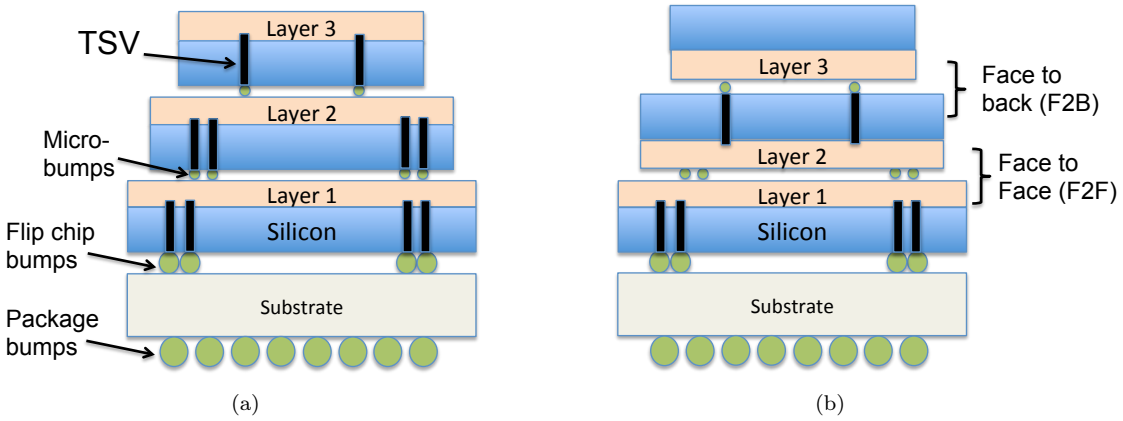


FIGURE 2.16: Staking techniques for 3DIC

Despite all its advantages and disadvantages, there are many applications using these architectures. CMOS (image sensors) imagers were the first industrial applications using 3D architecture [62]. Other candidates for 3D IC technology are memories and microprocessors (the stacking of memory on memory, memory on logic or memory on a processor). Besides the high performance applications, 3D IC is recognized as a key technology for heterogeneous products, for example in the areas of video games, mobile or in the digital world [63].

Current research conducted in term of development of technological processes for 3D IC are active worldwide. Xilinx has recently fabricated the Virtex 7 - 2000T family (as shown in the Fig 2.17) which comprises four FPGA layers horizontally connected by microbumps [64][2]. In the Virtex 7, TSVs are used for the connection with the package substrate. Some authors are interested in 3D FPGAs which are composed of many FPGA

layers stacked one above others. Although the 3D FPGA is still academic, many authors have strong believes that a 3D FPGA will be soon appearing in the near future. Virtex 7 is not so far from 3D FPGA while connecting horizontally many FPGA layers. A task placed on such 3D FPGA is a 3D task model that promises to have a minimal interconnection distance, then a minimal interconnection delay.

Other researchers conduct their research on a 3D multiprocessor core (3DCMP). The 3DCMP integrated multiple many core layers of 2D Mesh networks by connecting them with a vertical bus. This kind of networks is also called 3D Mesh networks as they give the possibility to communicate in the z-dimension [65].

Some 3D IC based multicore were presented and demonstrated, for example: the 3D-MAPs whose 64 custom cores are implemented with two-logic-die stack, the Centip3De which is based on ARM Cortex-M3 cores, etc. For a complete list of global actors (companies, laboratories, consortium, etc...) in the field of 3D integration (including packaging without the use of 3D TSV), the reader can referred to the reference [66].

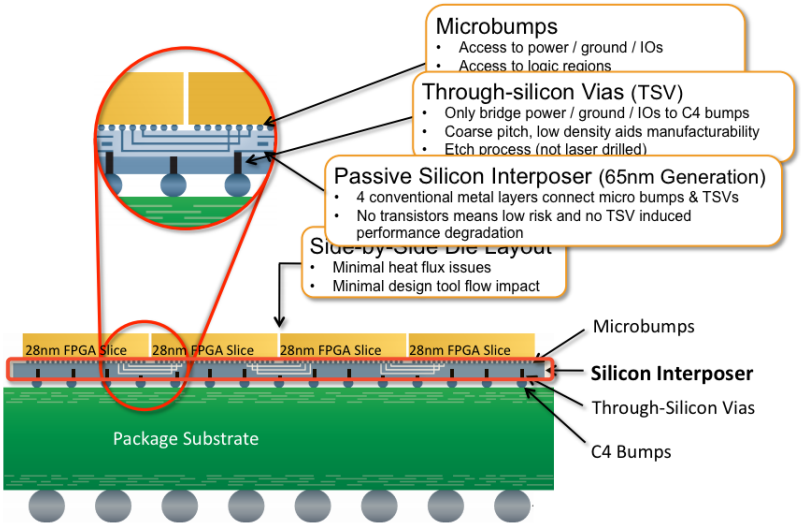


FIGURE 2.17: Virtex 7 - 2000T (figure taken from [2])

2.3.3 Task Scheduling and Placement in 3D Integrated Circuit

Due to their compact geometry, 3D integrated systems hold promises to offer many benefits: wire length reduction (which can provide two advantages: latency improvement and power reduction), high memory bandwidth improvement, coping with heterogeneous system and cost reduction.

However, this 3D technology faces some major challenges that hinder its industrial application. One of the main problems is the heat dissipation which can limit the operating frequencies and degrades the reliability of the chip. Other challenges include the definition of tools and methodologies, manufacturing costs, higher power densities from TSV or TSV overhead.

Several authors have proposed solutions to reduce the heat dissipation. In [67], the author discussed the solution to insert thermal vias in some areas or disperse thermal vias through the stacked layers to solve and optimize the problem of heat dissipation. Connecting a heat sink on the top of the 3D chip to extract the heat of the chip is also a solution that many researchers have studied. Another solution without modification of the hardware design of the chip is the task scheduling and placement. Scheduling and placing a task on different layers of the chip can have different thermal effects. For example, if a task is executed on the closest layer from the heat sink, the heat can be easier extracted. Based on this observation, Zhou et Zhang [68] propose an OS-level scheduling algorithm, named Balancing-by-stack, that performs thermal-aware task scheduling on a 3D chip multiprocessor (CMP). Li et al [69] also propose an OS level task scheduling algorithm, called TARS, to reduce the peak temperature on a 3D CMP. By considering two factors: frequencies assignment and inter-iteration data dependencies, TARS can achieve a 10°C reduction for the peak of temperature. Some other works addressing the thermal aspect on 3D core are [70][71].

Valero et al [72] present an efficient technique for managing hardware tasks as efficiently as possible on a 3D FPGA. Their algorithm is based on Vertex-List Structure which contains the candidate locations for the tasks. They propose Spatial-adjacency and Space-Time adjacency heuristic to select a certain location for each task and a metric to estimate the 3D fragmentation produced. However, these proposed algorithms may schedule and place tasks in a way that the currently executing tasks may block future tasks. As a result, the 3D FPGA will be fragmented again, some tasks can miss their deadlines and the overall execution time length will increase. Marconi et al [73][74] tried to solve this "blocking-effect" issue by proposing a blocking-aware heuristic called 4D Compaction. Their heuristic takes into account both the 3D spatial coordinates and time coordinate. It groups tasks with similar finish times together in order to form a larger free volume to future tasks, and it schedules the arriving tasks more compactly on a 3D FPGAs by avoiding "blocking-effect".

2.4 Summary

Almost existing scheduling and placement algorithms for the reconfigurable resources are related to the 1D or 2D homogeneous FPGA. In almost cases, authors consider that tasks are independent and their objective is to reduce the FPGA fragmentation, i.e. how to place a large number of independent tasks on a limited FPGA. However, for an application, tasks are dependent and need to exchange data between them to process their treatments. Moreover, recent FPGAs are heterogeneous while embedding different blocks as CLBs, BRAMs, DSPs, etc, which need a different placement strategy. Very few works address this heterogeneity during the task placement. Most of them predefine the feasible regions for each task of the application at design time, i.e. before the application gets started. This predefinition requires a big effort for the task designer to create a bitstream for each region of a task. Consequently, the system needs a big external memory to store all these bitstreams.

Then, for our knowledges, the research about the 3DRSoC architectures is still on progress and there is no existing spatio-temporal strategies for this kind of platform.

In our work, we propose first the scheduling strategies for hardware tasks executed in the 2D homogeneous FPGA (which includes the 2D Bloc Area FPGA and the 2D Free Area FPGA) and the 2D heterogeneous FPGA while considering the dependencies between tasks. We adapt the notion of hardware task relocation for these FPGAs, thus only one bitstream is needed for each task and the feasible regions for each task are founded at run-time by our proposed methods. Then, we extend these strategies to take into account the 3rd dimension of the 3D Homogeneous RSoC and the 3D Heterogeneous RSoC. For that, not only hardware tasks but also software tasks needs to be managed efficiently to exploit at maximum the advantages offered by these 3D architectures.

Chapter 3

Online spatio-temporal scheduling strategies for 2D homogeneous reconfigurable resources

As the execution flow of tasks is not known in advance in online scheduling, the decisions for arriving tasks should be taken at runtime. Tasks must be managed efficiently in time and space to exploit the advantages offered by the FPGA. A spatio-temporal scheduling algorithm needs to find the earliest starting time for arriving tasks (when) and the region to accommodate (where) them.

In this chapter, we propose different spatio-temporal scheduling strategies for dependent hardware tasks executing on two types of 2D homogeneous reconfigurable resources that are 2D Bloc Area and 2D Free Area FPGA. The objective of these strategies is to minimize the communication cost between tasks and to increase the FPGA utilization resource. Section 3.1 presents the first algorithm based on Pfair called Pfair Extension for Reconfigurable Resource (Pfair-ERR) that deals with online task scheduling and placement on 2D Bloc Area Model. Section 3.2 presents two algorithms based on Vertex List Structure (VLS) called the Vertex List Structure-Best Communication Fit (VLS-BCF) and Vertex List Structure-Best Hotspots Fit (VLS-BHF) that target 2D Free Area Model. Finally, we conclude this chapter with the summary section.

3.1 Pfair extension for 2D Bloc Area Model

3.1.1 Pfair for independent tasks

Considering a set of periodic independent tasks whose each task T_i is characterized by (Execution Time E_i, Period P_i, Deadline D_i). As mentioned earlier, Pfair is considered optimal for scheduling a set of periodic independent tasks on a multiprocessor system of N_p processors if the following constrain is respected :

$$\sum_{i=1}^{n} E_i/P_i \leq N_p$$

In Pfair [14] scheduling, a task T_i is breaking (or decomposed) into E_i "subtasks". We denote the k^{th} subtask of task T_i as T_i^k, where 1 $\leq$ k $\leq E_i$. Processor time is allocated in discrete time units or quanta, and we call a slot t the time interval $[t, t+1]$. The migrations and preemptions are allowed but not the parallelism for each task, i.e. a task may be allocated over time on different processors but not within the same slot. A schedule is defined by the sequence of allocation decisions over time.

Under Pfair algorithm, we associate with each subtask of T_i a pseudo-release $r(T_i^k)$ and a pseudo-deadline $d(T_i^k)$. The interval $[r(T_i^k), d(T_i^k)]$ is called a "window" of the subtask T_i^k and each subtask must be executed within this "window". These windows divide each period of a task into subintervals of approximately equal length. The deadline of the last window is the deadline of the task. $r(T_i^k)$ and $d(T_i^k)$ are calculated as following:

$$r(T_i^k) = \left\lfloor \frac{(k-1) * P_i}{E_i} \right\rfloor \text{ and } d(T_i^k) = \left\lceil \frac{k * P_i}{E_i} \right\rceil$$

Fig 3.1 shows a PFair example of three tasks T_1, T_2, T_3 executed on two parallel identical processors $Proc_1$ and $Proc_2$. These tasks are independent with their characteristics detailed in Table 3.1. The windows within each job are presented in Fig 3.1, for example T_1 has 3 windows $W_{1.1}$, $W_{1.2}$ and $W_{1.3}$, T_2 has 7 windows and T_3 has 4 windows.

We call L the list of ready subtasks in time t, given by:

$$L = \left\{ T_i^k | r\left(T_i^k\right) \leq t \text{ and } T_i^k \text{ not yet executed} \right\}$$

The priorities of subtasks in L are calculated based on an earliest-pseudo-deadline-first (EPDF) basis and several tie-breaking rules when multiple subtasks have the same deadline. In this thesis, we are not going to detail the tie-breaking rules but this information can be founded in [14].

If we assume that there are N_p possible processors, L^+ represents the ordered list of N_p highest priority subtasks of L. The elements of L^+ are subtasks that must be executed at time t or, in other words, subtasks that must be assigned to the N_p processors. At each slot time t, Pfair algorithm executes the following steps:

- Establish the list of ready subtasks L
- Build L^+ by extracting the N_p first elements of L
- Assign the subtasks of L^+ to the N_p processors

	Task 1	Task 2	Task 3
Execution E_i	3	7	4
Period P_i	6	10	12
Deadline D_i	6	10	12

TABLE 3.1: Characteristics of tasks

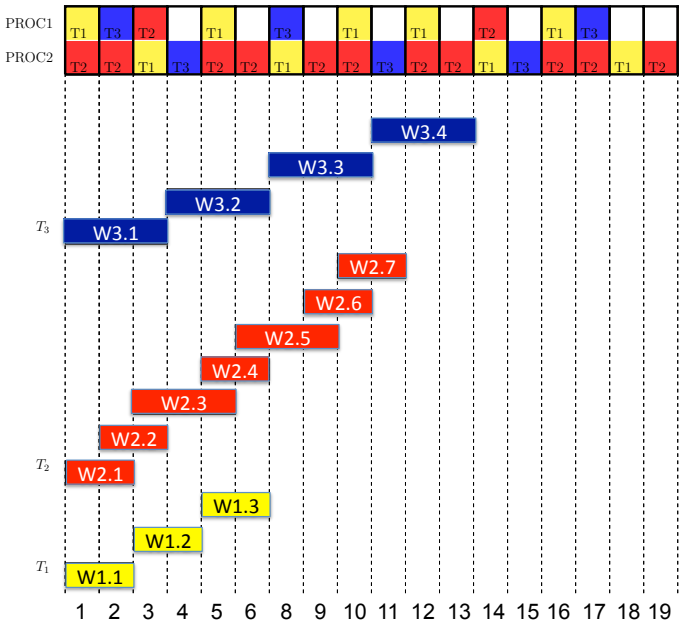


FIGURE 3.1: Assignment of T_1, T_2, T_3 to two parallel processors

We reveal that there are many preemptions and migrations during the task scheduling. In the worst case, a task may be migrated at each time it is scheduled. For example,

at slot time 3, instead of assigning the subtask of T_2 on $Proc_1$ and the subtask of T_1 on $Proc_2$, the algorithm can totally reverse the assignment of T_2 and T_1 so that the migration doesn't occur at this time. To limit this problem, Dalia et al [75] propose some heuristics to avoid the migrations of tasks. Some other heuristics have also been developed to avoid the number of preemptions as Dp-Fair [15] or Bfair [16].

The Pfair classic is used for independent tasks executed on a MPSoC system. However, in an application, tasks must be dependent and exchange data between them. In the next sections, we propose to adapt Pfair for the reconfigurable resource while taking the dependencies between tasks. The communication cost between tasks and also the number of migrations and preemptions will be evaluated in order to analyze the efficiency of the adapted method on the reconfigurable resource.

3.1.2 2D Bloc Area scheduling and placement problems

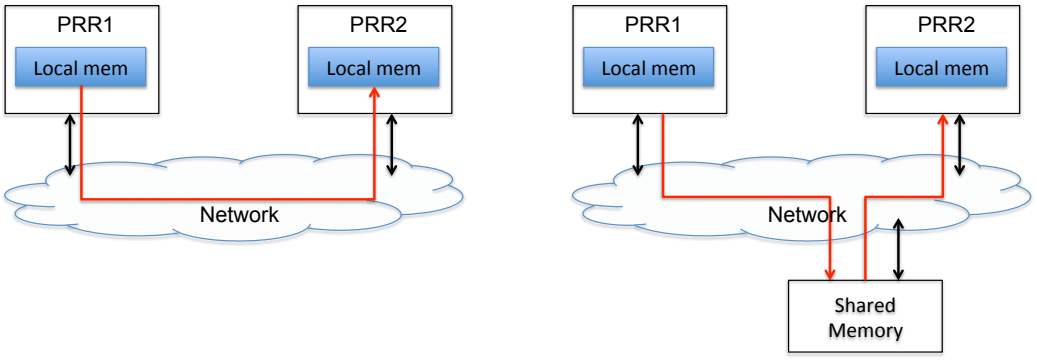
In a 2D Bloc Area model, each bloc can accommodate at most one task at a time. In the cases that tasks are independent, the most difficult comes from the scheduling of tasks but not the placement. However, for dependent tasks, the placement plays also an important role as it can affect the communication delay between tasks and consequently increase the overall execution time of the application. The more the distance between communicating tasks is far and the more the amount of exchanged data between two tasks is large, the more the communication cost is penalized. Furthermore, interconnection of two tasks onto the reconfigurable resources consumes logic elements and routing signals. Therefore, reducing the communication cost not only simplifies the place and route of the tasks onto this layer but also reduces the energy consumption.

To solve this problem of spatio-temporal scheduling, we propose to combine the Pfair scheduling algorithm and the Nearest Neighbor Possible (NNP) while considering the data dependencies between tasks. As Pfair is known as an optimal algorithm for scheduling independent tasks on multiprocessor system, we extend it to consider the task dependencies. Then we adopt NNP technique to place communicating tasks as close as possible to reduce the communication cost.

3.1.3 Assumption

- We consider an architecture with a set of Partial Reconfigurable Regions (PRRs) and a global memory connected by a network-on-chip. Similar to multiprocessor systems, we suppose that each PRR contains local memory to store data before sending them to another task, and to store data when receiving data from another task. The global shared memory is supposed to be used with UMA technique.

- The communication between two tasks is supposed to be done between the last subtask of the sender task and the first subtask of the receiver task. We distinguish two types of communication: the direct communication and the communication via the shared memory. When the direct communication is chosen (Fig 3.2(a)) the data is sent directly from the local memory of the sender task to the local memory of the receiver task. On the other hand, when the communication via the shared memory is chosen (Fig 3.2(b)), the data of sender task are written to the shared memory and the receiver task accesses the shared memory where the data are stored, to bring back the data.



(a) Communication in direct (b) Communication via the shared memory

FIGURE 3.2: Two possible types of communication between two communicating tasks

- For the direct communication, if two communicating tasks are sequentially executed on the same PRR, no transfer is necessary between two tasks and the communication cost is considered to equal to zero. This type of communication needs a specific management of send and receive operations, by ensuring that the sender and receiver tasks read and write data in the same internal memory at the same addresses. On the other hand, if two tasks are executed on different PRRs, the

direct communication cost is calculated based on the distance between two PRRs and the amount of exchanged data between two tasks.

- For the communication via the shared memory, two operations of variables/messages are needed : write operation for the sender task and read operation for the receiver task. As the UMA technique is used for the shared memory, the physical locations of tasks do not impact the performance. We suppose that the communication cost via memory is two times the amount of exchanged data between two tasks.

3.1.4 Formulation of the communication problem

Let G the task graph of the application, G is defined as

$$G = \{T, C\} \tag{3.1}$$

with T the set of application tasks and C the set of communications between tasks. The set of tasks is defined by

$$T \quad = \quad \{T_i\} = \{(T_i^k)\} \qquad \forall\, i = 1, \ldots, N_T \text{ and } \forall\, k = 1, \ldots, E_i \tag{3.2}$$

with N_T the number of tasks of the application. The communications between tasks are defined by

$$C = \{C_{i,j}\} \qquad \forall\, (i,j) = (1, \ldots, N_T), (1, \ldots, N_T) \tag{3.3}$$

We define $D_{m,n}$ as the Mahattan distance between PRR_m and PRR_n. This distance is defined by

$$D_{m,n} = |Px_m - Px_n| + |Py_m - Py_n| \tag{3.4}$$

with Px_m and Py_m the physical barycenter position of the partial reconfigurable region PRR_m on the reconfigurable layer.

The communication between two tasks is supposed to be done between the last subtask of the sender task and the first subtask of the receiver task. Therefore, at the time when

the last subtask of the sender task finishes its execution, we need to find the allocation for the first subtask of the receiver task in order to minimize the communication cost between them. Then the problem consists in finding $A_{j,n}$ of the first subtask T_j^1 on PRR_n to minimize the communication cost with the instantiation $A_{i,m}$ of the last subtask $T_i^{E_i}$ on PRR_m. Hence, the communication cost between two tasks T_i and T_j can be written as

$$CommCost_{i,j} = \begin{cases} 0 & \text{if } \exists \text{ n} = \text{m} \mid A_{j,n} = A_{i,m} = 1 \\ \alpha * D_{m,n} * C_{i,j} & \text{if } \exists \text{ n} \neq \text{m} \mid A_{j,n} = A_{i,m} = 1 \\ 2 * \beta * C_{i,j} & \text{if } \nexists \text{ n} \mid A_{j,n} = 1 \end{cases} \tag{3.5}$$

with α a time constant delay which models the direct communication on reconfigurable layer which only depends on the distance between the two PRRs, β a time constant delay which models the communication via shared memory. The communication cost is equal to 0 in the case two communicating tasks sequentially execute on the same PRR.

Reducing the communication cost between tasks leads to reduce the global communication cost. We consider the global communication cost as the sum of the total direct communication cost and the total memory communication cost. Thus, the minimization of global communication cost for the complete application can be written as

$$Min \sum_{i=1}^{N_T} \sum_{j=1}^{N_T} (CommCost_{i,j}) \tag{3.6}$$

3.1.5 Pfair extension algorithm for Reconfigurable Resource (Pfair-ERR)

From the decomposition of tasks into subtasks given by Pfair, we suppose that the communication between two tasks is done between the last subtask of the sender task and the first subtask of the receiver task (for example in Fig 3.3). Compare to the classical Pfair algorithm which all tasks are independent, this communication model takes into account the dependencies of tasks and modifies the arrival time of receiver tasks.

The main strategy is to favor the direct communications to minimize the amount of data stored in the memory. At the slot time t when several direct communications may occur, our algorithm evaluates the one with a large amount of data first. The algorithm tries to

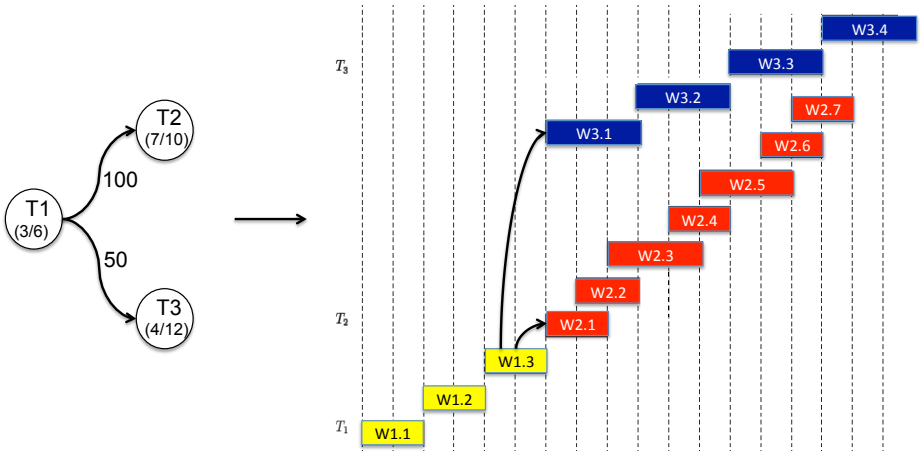


FIGURE 3.3: Assignment of T1, T2, T3 to two parallel PRRs

assign the first subtask of the receiver task on the same PRR as the last subtask of the sender task when possible. In this case, no transfer is needed between the two tasks and the communication cost can be considered equal to zero. If this assignment is not possible, the first subtask of the receiver task will be assigned on the nearest available PRR from the last subtask of the sender task so that the direct communication is reduced. Then, if all direct communications cannot be ensured at that time, the memory communication will be chosen.

Beside the main objective of our algorithm which is to minimize the communication cost between tasks, we integrated also a technique called Early-release fair [76] (ERfair) to maximize the processor use and reduce the execution time costs. Under ERfair, if two subtasks are part of the same task, then second subtask becomes eligible for execution as soon as the first completes. As a result, no processor is ever idle while there exists uncompleted jobs to schedule.

3.1.6 Evaluation

This section presents some results produced by our algorithm Pfair-ERR. In order to evaluate the quality of these results, we compare Pfair-ERR with an offline recursive called Branch and Bound (BB). The BB technique consists in producing all non-preemptive and non-migrating scheduling solutions.

Tasks graph					BB-Dir(1)			BB-Mem(2)			The proposed Algorithm(3)					Gain: (3)/(1)		Gain: (3)/(2)	
N_T	**Dep**	**E**	**ExData**	**Sol**	**Dir**	**Mem**	**Te**	**Dir**	**Mem**	**Te**	**Dir**	**Mem**	**Te**	**M**	**P**	**%Com**	**%Te**	**%Com**	**%Te**
5	4	23	157	96	107	0	11	107	0	11	107	0	11	0	0	0.000	0.000	0.000	0.000
5	6	23	227	384	49	140	16	49	140	16	49	140	16	0	0	0.000	0.000	0.000	0.000
6	5	27	235	576	107	130	14	176	30	12	176	30	11	3	3	0.131	0.214	0.000	0.083
6	6	27	288	288	176	136	11	176	136	11	176	136	11	0	0	0.000	0.000	0.000	0.000
7	8	28	363	7306	107	386	15	211	208	16	188	128	13	3	3	0.359	0.133	0.246	0.188
7	8	28	404	2016	135	440	12	176	368	12	176	368	10	3	2	0.054	0.167	0.000	0.167
7	9	28	435	2016	135	502	12	176	430	12	176	430	10	3	2	0.049	0.167	0.000	0.167
7	10	28	462	10237	107	584	15	211	494	16	331	128	13	3	3	0.336	0.133	0.349	0.188
8	7	32	376	288	53	398	11	84	336	10	118	268	10	1	1	0.144	0.091	0.081	0.000
8	8	32	385	1732	53	346	15	113	284	14	152	216	14	1	2	0.078	0.067	0.073	0.000
8	10	32	469	1732	53	514	15	113	452	14	152	384	14	1	2	0.055	0.067	0.051	0.000
9	8	35	428	1152	53	502	10	84	440	11	204	268	10	3	6	0.150	0.000	0.099	0.091
9	9	35	457	6651	53	678	14	260	358	14	233	134	13	4	5	0.498	0.071	0.406	0.071
10	9	39	501	4032	19	714	12	46	660	11	295	268	11	4	6	0.232	0.083	0.203	0.000
10	11	49	528	4752	45	496	19	98	480	18	84	410	18	1	1	0.087	0.053	0.145	0.000
													Average			**0.145**	**0.083**	**0.110**	**0.064**

TABLE 3.2: Assignment of tasks onto 4 PRRs. N_T : Total number of tasks, **Dep**: Total number of dependencies, i.e the total number of edges in the task graph, **E**: the sum of execution cost of all the tasks, **ExData** : Total number of exchanged data between tasks, **Sol**: total number of possible solutions generated by BB, **BB-Dir**: non-preemptive and non-migrating solution with the smallest direct communication cost (**Dir**), **BB-Mem**: non-preemptive and non-migrating solution with the smallest memory communication cost (**Mem**), **Te**: Total execution time, **%Com**: Gain in term of total communication cost between our algorithm and BB, **%Te**: Gain in term of total execution time between our algorithm and BB, **M**: Total number of migrations due to our algorithm, **P**: Total number of preemptions due to our algorithm

The table 3.2 shows the results for 15 sets of tasks executing onto 4 PRRs architecture. Each set is a DAG and we suppose that except the source task, each task in the set must have at least one predecessor. The characteristic of a set is shown by four parameters (N_T, Dep, E, $ExData$) on the left of the table. The BB-Dir represents the non-preemptive and non-migrating solution with the smallest direct communication cost. The BB-Mem represents the non-preemptive and non-migrating solution with the smallest memory communication cost. The global communication cost is calculated according to the equation 3.6 whose the parameters α and β are set to 1. For every set, our Pfair-ERR always produces a solution having a shorter or equal overall execution time (noted Te) and a less expensive global communication cost (GlobCom) than BB-Dir and BB-Mem. Pfair allows the migrations and preemptions of tasks, therefore when one subtask finishes its execution and needs to send data in direct, other subtasks in progress which don't send data at that time, could be stopped momentarily and will be resumed after the direct communication is finished. Thus, direct communications are favored, and this results in a reduction of memory communication cost. Furtermore, the overall execution time is also reduced by exploiting all the capacity of processors. On average, Pfair-ERR shows up to 14.5% reduction in global communication cost compared to BB-Dir and 11% compared to BB-Mem. In terms of the overall execution time, Pfair-ERR allows 8.3% reduction compared with BB-Dir and 6.4% reduction compared with BB-Mem.

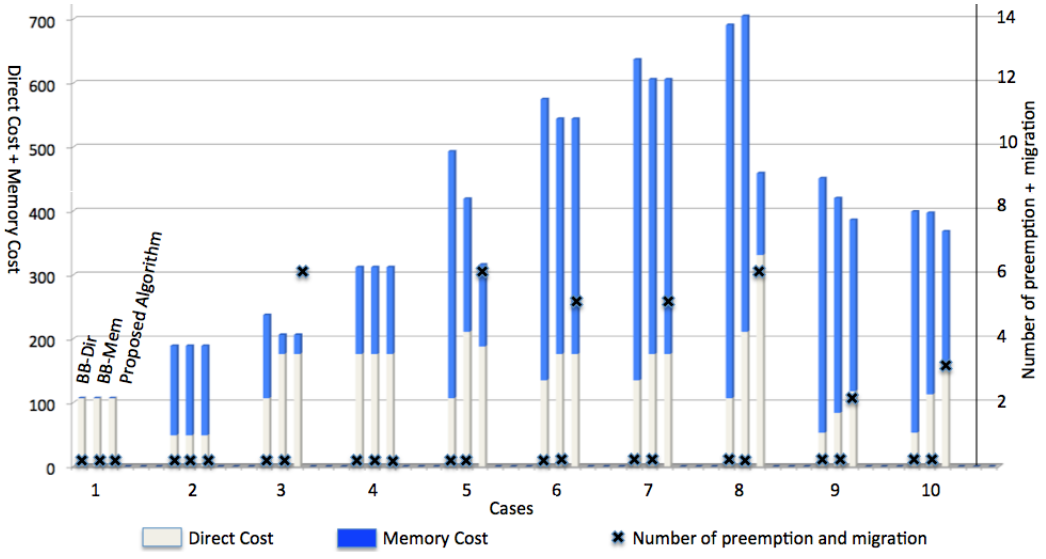


FIGURE 3.4: Comparison of different solutions in term of total communication cost and total number of preemptions and migrations

Even Pfair-ERR brings benefits in terms of the global communication cost, we could not ignore the number of migrations and preemptions that may be very costly in the

reconfigurable architecture. Fig 3.4 analyzes the global communication cost and the total number of migrations and preemptions of the first ten cases of the Table 5.1. The memory communication cost produces by our proposal is regularly less than or equal to the equivalent solutions of BB, which leads to a lower (or equal) global communication cost. When there is no migration and preemption, we obtain an equivalent scheduling relative to the BB-Dir and BB-Mem. In these cases, the global communication cost produced by Pfair-ERR is the same as BB-Dir and BB-Mem (cases 1,2 and 4 on Fig 3.4). However, for the remaining cases, we note a large number of preemptions and migrations of Pfair's sub-tasks caused by our algorithm. Preemptions and migrations require saves of context in order to be able to resume the task later without any data loss. They require non negligible costs, therefore it is also necessary to take into account of these costs during the task scheduling.

The Pfair-ERR aims to propose a strategy to schedule and assign tasks in order to minimize the communication cost between them so that the global communication cost is reduced. Our proposed communication cost model allows evaluating the communication cost in direct and via the shared memory. By varying the α and β parameters, the Pfair-ERR can be adapted to different 2D Bloc Area architectures with different technologies. The cost caused by preemptions and migrations is out of our scope. However, in order to make a fair comparison, this cost should be taken into account. Then, the efficient of our algorithm can be truly evaluated depending on this cost. We did not go further to analyze how much this cost can be. For some applications, preemptions and migrations are allowed but for some others, they should absolutely not used. For the reminder of this thesis, we propose other techniques where the preemptions and migrations are not allowed.

3.2 VertexList extension for 2D Free Area Model

3.2.1 2D Free Area Model and Task Model

A 2D Free Area Model is defined by a rectangle of W_{FPGA} * H_{FPGA} homogeneous Configurable Logic Blocks (CLBs). We suppose that the benchmark for each task T_i in an application is synthesized with the corresponding tools. After this synthesis, the following information is extracted for each task in term of: reconfiguration time HR_i, worst

case execution time HE_i, the required resources and the position for the Input/Output interface of the task. The required resources of a task T_i are defined as a rectangle of w_i*h_i CLBs. The I/O interface is supposed to be located on the top left CLB of the task. These I/Os are connected to a communication mesh allowing inter-task communications, like dyNoC [77].

An application is defined as a DAG whose edges correspond to the communication between tasks. Each task T_i is defined by

$$T_i = \{w_i, h_i, HR_i, HE_i\}$$

An example of implemented tasks which are synthesized with the tools targeting the Virtex-4 XC4VLX200, is given in [4]. The information obtained for task implementation on Virtex-4 XC4VLX200 ($W_{FPGA} = 116$ and $H_{FPGA} = 192$) are :

No	Hardware Tasks	$w_i(CLBs)$	$h_i(CLBs)$	$HR_i(ns)$	$HE_i(ns)$
1	funtionPOWER	14	32	252560	43183
2	*adpcm_decode*	10	32	180400	770302
3	*adpcm_encode*	10	32	180400	1031213
4	FIR	33	32	595320	1565980
5	*mdct_bitreverse*	32	64	1136520	449412
6	mmul	25	64	892980	57278

TABLE 3.3: Examples of implemented hardware tasks in [4]

Placing, in real-time, a maximum number of tasks within the limited area is really challenging. Scheduling and placement of tasks on the 2D Free Area FPGA consists in determining, at run time, the arrival time of tasks (temporal point) and the regions the tasks should be placed (spatial point). The arrival time of tasks depends on its dependencies with other tasks and the placement of tasks is computed based on the current state of running tasks on the device. A rectangular region $R_{k,i}$ is called a feasible region for T_i if and only if all the resources in this region are available and the total number of resources in this region must be equal to the required resources of T_i. We define $R_{k,i}$ as following

$$R_{k,i} = \{x_k, y_k, w_i, h_i\}$$

where (x_k, y_k) is the coordinates of the bottom left point of the region on the FPGA, (w_i, h_i) is the width and the height of the region.

To simplify, finding a region to place a task T_i consists in finding a feasible point (x_k, y_k) on the FPGA to place the task so that there is no overlapping i) with other tasks currently running and ii) with the FPGA border.

3.2.2 Vertex List Structure (VLS)

We introduce here some main views about Vertex List Structure (VLS) whose details are given in [45]. As mentioned in the background section, the VLS is used to geometrically represent all the free area boundaries of the FPGA. During task placement, the FPGA can be divided in one or several holes. The VLS contains a set of different Vertex Lists (VLs), each VL represents all the vertices in an FPGA free holes. The initial VLS contains only one VL with four points, which represent the four corners of the reconfigurable resource (Fig 3.5). When a task is placed in one specific point of the VL, the VLS is updated by removing this point and by adding the others points generated by the placement. When a task must be removed, the VLS is also updated.

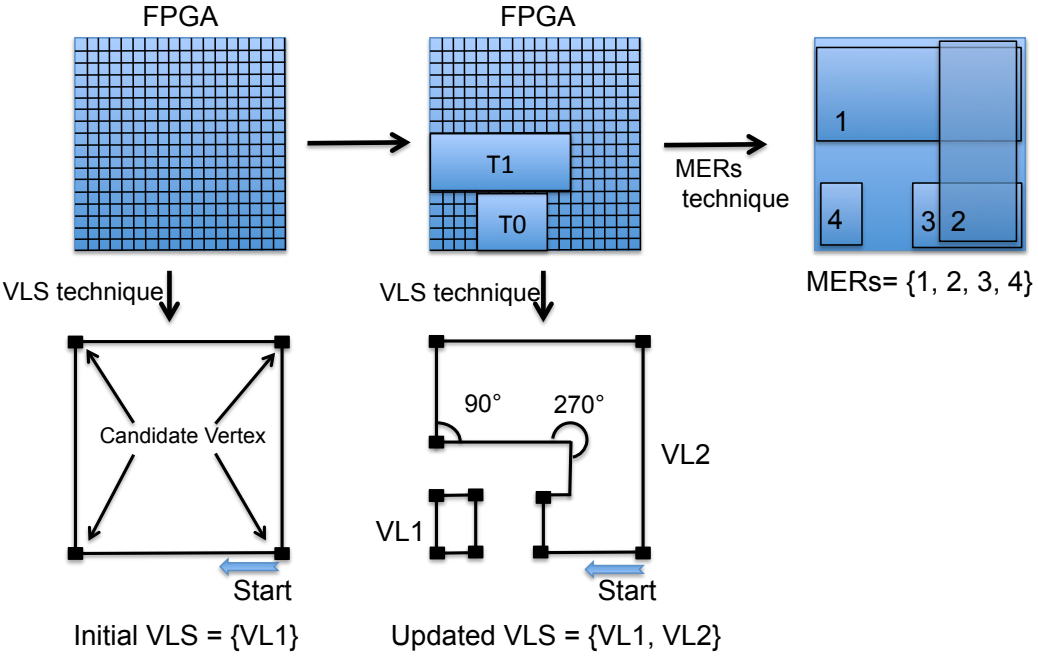


FIGURE 3.5: VLS and MERs techniques

Two types of vertex can appear in VLS as shown in Fig 3.5: 90 degree-angle vertex and 270 degree-angle vertex. However, only the first type (90 degrees) is considered as candidate vertex for task placement due to the lower value of fragmentation compared to another type [45]. For each new arriving task, the algorithm starts at bottom right vertex

of the VL, then travels clockwise and looks for a feasible position to insert task at all the vertices marked as candidate. A VL is fully travelled when the Start Vertex is found again. According to the chosen criteria, the new task will be placed on the corresponding candidate vertex.

3.2.3 Communication cost and hotspot problem

For our knowledges, most of previous works address the spatial but not the temporal problem. They consider that tasks are independent and simplify the problem to the placement of boxes on a rectangular surface without considering the communication between tasks. Most of them propose the techniques to reduce the task rejection and the fragmentation rate, thus increase the FPGA utilization. However, their hypothesis is not realistic and hides the difficulties to route the data between tasks and the need of memory to store the exchanged data between two tasks if needed.

The problem is more complex if the dependencies between tasks are considered. Rather than constrain the definition of a very efficient NoC to ensure the fast communication, we claim that it is important to include the communication criteria directly in the spatio-temporal scheduling. Communicating through the shared memory cannot guarantee the high-performance and high-bandwidth requirements for running tasks on the FPGA. Therefore, we rely on direct communication cost. The long communication path between task directly leads to the high latency and energy consumption. Furthermore, placing communicating tasks far apart may increase the number of hotspots and network loads when several communications exist simultaneously on the circuit. In this section, we propose two algorithms whose details will be given in the next. The first algorithm, called Vertex-List Best Communication Fit (VLS-BCF) places arriving tasks so that the direct communication cost between communicating tasks is minimized. The second algorithm, called Vertex-List Best HotSpot Fit (VLS-BHF) places arriving tasks so that the number of hotspots appearing on the circuit is minimized and the direct communication cost is low.

3.2.4 Definition and Assumption

We introduce here some definitions and assumptions that we integrated in our algorithm:

- A task T_i is said ready to be scheduled at time point t when and only when all predecessors of this task have already finished their executions at that time.

- The communications between tasks are supposed to be done at the end of transmitter task execution and at the beginning of receiver task execution. We defined two possibilities to support these communications: i) the direct communication and ii) the communication via a shared memory located outside the FPGA. A task must have five following stages: ReceiveData→ Config → Execute → SendData → Exit.

- A communication between T_i and T_j is said ready at time t if and only if T_i has finished its execution and T_j is ready to be scheduled at this time.

- A communication between T_i and T_j is said possible in direct at time t if and only if T_i has finished its execution, T_j is ready to be scheduled and a feasible region on the FPGA is available to accommodate T_j at this time.

- A communication between T_i and T_j is said not possible in direct at time t if T_i finishes its execution but T_j is still not ready due to the data dependencies with its non-accomplished predecessors or due to the inadequate space on the FPGA at this time. In this case, the data produced by T_i are stored into the shared memory and will be retrieved by T_j later.

- We suppose that when two tasks exchange data in direct, a communication segment connecting these two tasks illustrates the direct communication between them. Several communications can occur at the same time, the number of hotspots created on the circuit are supposed to be related to the number of intersection points between communication segments. The figure on the top right of Fig 3.8 shows an example that a hotspot point is created due to the intersection between two communication segments : (T_2 - T_4) and (T_3 - T_5).

- To ensure all communications between tasks in the application, we suppose that the deadline execution time for the task is large enough so that no task is rejected (soft real-time). Thus, when one task T_i is requested to be scheduled but cannot be placed due to the inadequate space on the FPGA at this time, it can still be placed later.

3.2.5 Formalization

Similar to the previous section, we assume that when two tasks exchange data in direct, the communication between them is directly linked with the Manhattan distance between their I/O interfaces. This distance is the sum of the absolute differences of two I/O coordinates. As the I/O interface is located on the top left of a task, for a task T_i placing at (x_m, y_m) on the FPGA, its I/O interface will be located at (x_m, $y_m + h_i$). Then, the distance, noted $D_{mi,nj}$, between a task T_i placing at (x_m, y_m) and a task T_j placing at (x_n, y_n) is computed by the following expression

$$D_{mi,nj} = |x_m - x_n| + |y_m + h_i - y_n - h_j| \tag{3.7}$$

The communication cost $CommCost_{i,j}$ between two tasks T_i and T_j can be written as

$$CommCost_{i,j} = \begin{cases} \alpha * D_{mi,nj} * C_{i,j} \\ 2 * \beta * C_{i,j} \end{cases} \tag{3.8}$$

where α defines a constant delay which models the direct communication on reconfigurable layer, β a time constant delay which models the communication via shared memory and $C_{i,j}$ defines the amount of exchanged data between T_i and T_j.

The problem consists in choosing the vertex (x_m, y_m) to place T_i and (x_n, y_n) to place T_j so that the communication cost between two communicating tasks is minimized. Hence, the minimization of the global communication cost for the complete application can be written as

$$Min \sum_{i=1}^{N_T} \sum_{j=1}^{N_T} (CommCost_{i,j}) \tag{3.9}$$

The results in the section 3.1.6 showed that by favoring the direct communications, we can avoid the memory communications which are costly in terms of latency. Consequently, the global communication cost will be reduced. Therefore, the problem of minimizing the global communication cost can be simplified as the problem of minimizing the total direct communication cost.

3.2.6 Vertex List Structure - Best Communication Fit (VLS-BCF)

The main objective of VLS-BCF is to determine, during the spatio-temporal scheduling, the best vertices for new arriving tasks to reduce the total direct communication cost. At each time when a (or several) task(s) finishes its (their) execution, VLS-BCF realizes three following steps :

- Computation of the list, LC, which contains the "ready" communications. Each communication is represented as $\{T_i \rightarrow T_j$: amount of exchanged data$\}$
- Sorting the LC list in descending order of communication priority. This priority is calculated based on: i) the type of possible communications: direct communication or memory communication ; ii) the amount of exchanged data in each communication. In our algorithm, the direct communications exchanging more data have a higher priority.
- For each communication in LC which can be ensured in direct, the algorithm tries to find the best vertex for the receiver task. The placement of receiver task on this best vertex must guarantee that the Manhattan distance between the I/O interface of transmitter task and the one of receiver task is shortest among all vertices of VLS. If the receiver task cannot be placed at this time due to inadequate space in the FPGA, the communications related to the receiver task is pushed in waiting list for the next schedule time.

By limiting the long communications with large amount of data, we claim that the objective described in 3.9 is achieved.

3.2.7 Example of VLS-BCF

To better understand the proposed algorithm, we present an example of a set comprising 8 tasks (Fig 3.6) which must be scheduled and placed on a 2D Free Area Model. The values between parentheses in the nodes represent the time needed to reconfigure and execute the task. The edges with numbers represent the amount of exchanged data between tasks.

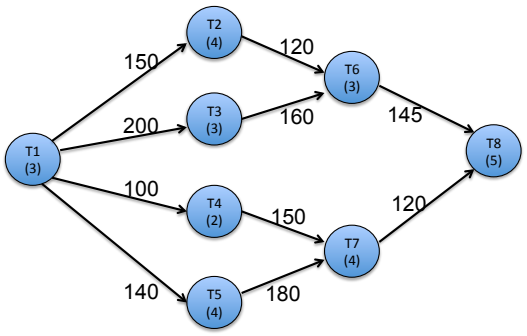


FIGURE 3.6: Set of 8 dependent tasks

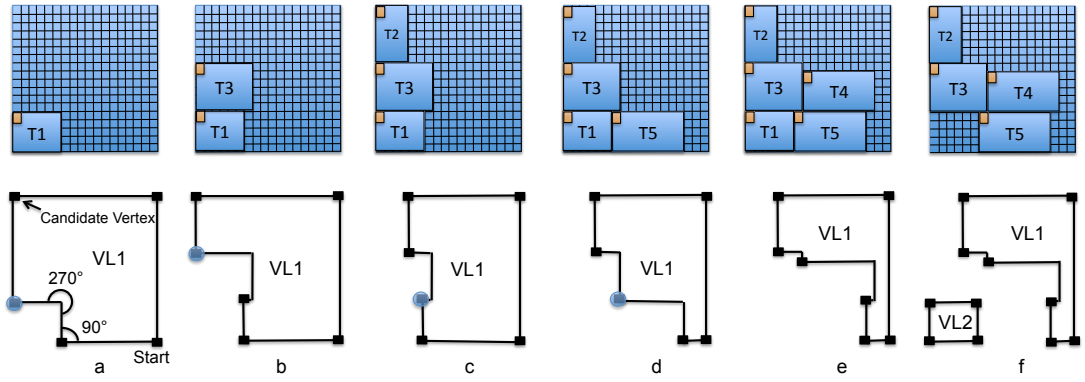


FIGURE 3.7: Evolution of FPGA status and VLS at each insertion or extraction of task by using VLS-BCF strategy

The obtained placement is shown in Fig 3.7. In this figure, only four first communications of the first schedule time are presented. At the beginning, T_1 is placed at the Bottom Left corner of the FPGA and VLS is updated as in Fig 3.7(a). At the end of T_1 execution, a list of communication LC is built, then sorted in descending order of number of exchanged data by T_1: ($\{T_1 \rightarrow T_3$:200$\}$, $\{T_1 \rightarrow T_2$:150$\}$, $\{T_1 \rightarrow T_5$:140$\}$, $\{T_1 \rightarrow T_4$:100$\}$). Thus, T_3 is chosen to be placed first. Among all candidate vertices, the vertex with a small blue circle in Fig 3.7(a) gives the smallest distance between the I/O of T_1 and the I/O of T_3. Therefore, T_3 is placed at this vertex and VLS is updated as in Fig 3.7(b).

After the placement of T_3, the algorithm decides to place T_2 at the vertex with a small blue circle in Fig 3.7(b) because it gives the smallest distance between the I/O of T_1 and the I/O of T_2. VLS is then updated as in Fig 3.7(c). Following the same strategy, T_5 and T_4 are placed as presented in Fig 3.7(d) and Fig 3.7(e). When all the communications listed in LC are done, task T_1 is removed from FPGA (Fig 3.7(f)). In this case, VLS is divided into two VLs: VL_1 and VL_2.

3.2.8 Vertex List Structure - Best HotSpot Fit (VLS-BHF)

We assume that by limiting the long communications with large amount of data, the number of hotspots appearing in the FPGA can be reduced. However, this number can be further reduced by applying another strategy during the task placement. Instead of placing an arriving task in order to reduce the direct communication cost, we can place the task so that there are fewest intersections between communication segments. When several vertices can satisfy this fewest intersection condition, the one creating less direct communication cost is chosen.

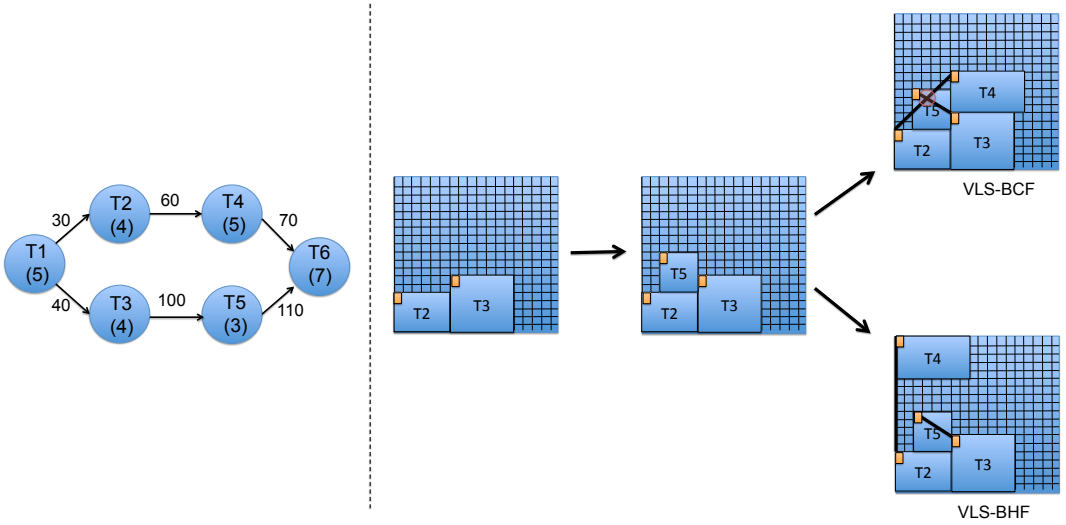


FIGURE 3.8: Evolution of FPGA status and VLS at each insertion or extraction of task by using VLS-BHF strategy

An example for VLS-BHF is given in Fig 3.8. According to the task graph, two communications $\{T_2 \rightarrow T_4 = 60\}$ and $\{T_3 \rightarrow T_5 = 100\}$ must occur at the same time. VLS-BHF privileges to place T_5 before T_4 because the amount of data consumed by T_5 is higher. No current communication exists at the time T_5 is requested, thus T_5 is placed near by T_3. The communication between T_3 and T_5 is represented by a communication segment. Then, when T_4 is requested to be placed, VLS-BCF places T_4 so that the communication distance from T_2 to T_4 is minimized. However, without information about the network structure which supports the communication and due to our simplified model of routing between tasks, this placement of T_4 could probably create a hotspot point due to the intersection between communication segments. Different to VLS-BCF, VLS-BHF places T_4 in the top left. In this case, there is no intersection point and the distance from T_2 to T_4 is also limited.

3.2.9 Evaluation

This section presents some results produced by our proposed methods called **VLS-BCF** and **VLS-BHF**. In order to evaluate the quality of these results, we compare our methods with some others in terms of the total direct communication cost and the total number of hotspots. Others comparing methods are the following:

- **Mer-BottomLeft** keeps all MERs at each schedule time and places the new task on the Bottom Left point of the rectangle having the smallest y-coordinate of the BL point ;
- **Mer-BestArea** places the new task into the smallest area rectangle which can receive the task ;
- **Mer-Contact** places the new task into a position where the length of the perimeter of this task is maximum touched with the FPGA edge or with the previously placed tasks ;
- **Mer-BestLong** places the new task into the rectangle that $max\{w_r - w_i; h_r - h_i\}$ is smallest, with w_r, h_r the width and height of the free rectangle and w_i, h_i the width and height of the new task T_i ;
- **Mer-BestShort** places the new task into the rectangle that $min\{w_r - w_i; h_r - h_i\}$ is smallest. The details of these five MERs methods can be found in [39] [78] ;
- Our method is also compared with the Vertex List Structure - First Communication Fit (**VLS-FCF**) which runs clockwise from the Start Vertex and tries to place the new task at the first found Vertex that task can be fitted.

For our comparisons, we randomly generate different sets of tasks. Each set starts with a source task and finishes by a sink task. The dimension of the FPGA, the characteristics of each set and also of each task T_i in the set are given in table 3.4. Fig 3.9 shows the comparison in terms of the total communication cost and Fig 3.10 shows the comparison in terms of hotspots for 6 sets of tasks placed on a $(400 * 400)$ FPGA.

Comparison of VLS-BCF with MER-BottomLeft, MER-BestArea, MER-BestContact, MER-BestLong, MER-BestShort and VLS-FCF

W_{FPGA}	H_{FPGA}	N_T	$C_{i,j}$	w_i	h_i	$HR_i + HE_i$
400 CLBs	400 CLBs	[6-30] tasks	[10-1000] data	[20-100] CLBs	[20-100] CLBs	[2-7] time units

TABLE 3.4: FPGA dimension and task characteristics

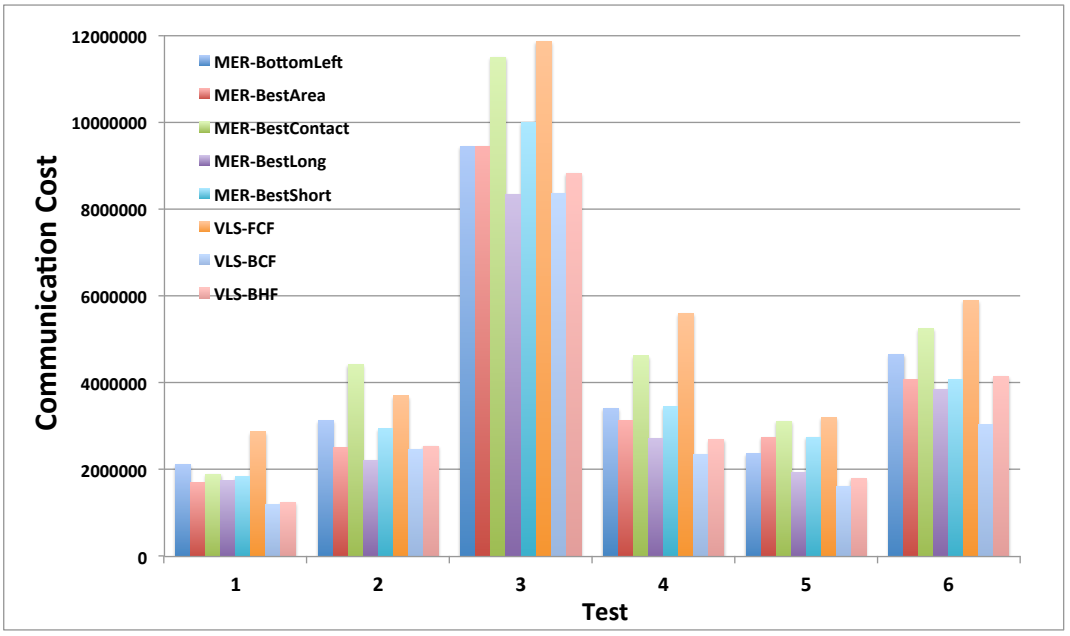


FIGURE 3.9: Comparisons of direct communication cost for different scheduling and placement techniques

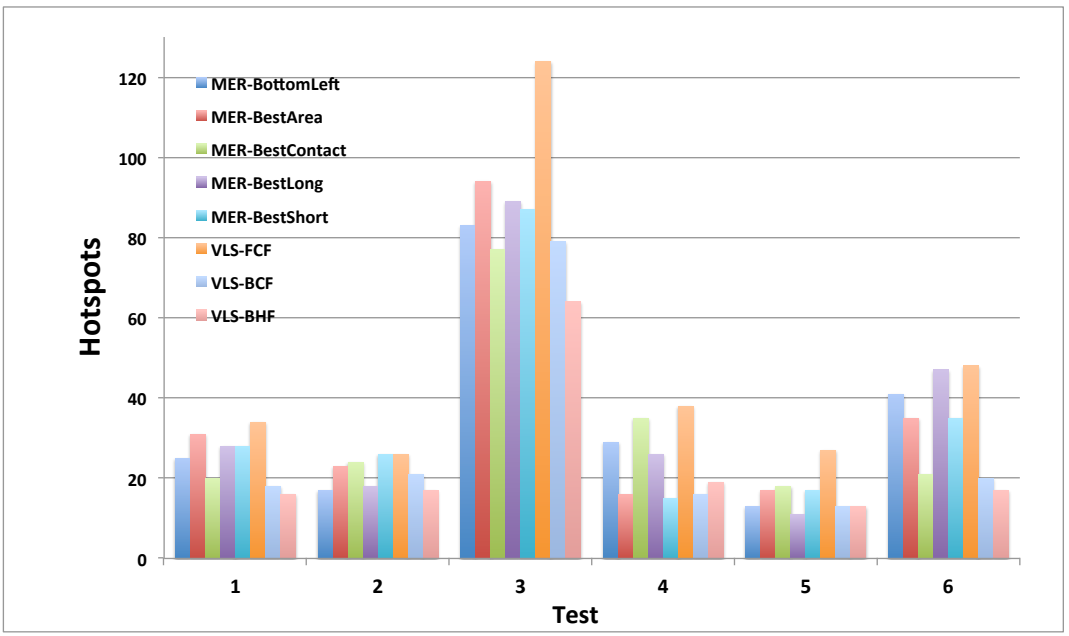


FIGURE 3.10: Comparisons of hotspots for different scheduling and placement techniques

In Fig 3.9, we observe that the direct communication cost our method VLS-BCF is regularly less than other methods. The communication cost produced by our method is reduced on average by 11,5% compared with the minimum communication cost among all the comparing methods and by 48% compared with the maximum communication cost. Obviously, in some cases, for example case 2, the communication cost produced by VLS-BCF is slightly bigger than Mer-BestLong. This difference can be explained by the fact that the numbers of vertices created in MERs methods are much higher than in VLS method and some corners of MERs are not considered in the VLS. Mer-BestLong leads to more placement choices for the tasks. Thus, in some cases, placing an arriving task on one of these MERs corners can give a smaller communication distance for the task.

The main objective of VLS-BCF focuses on the reduction of the global direct communication cost between tasks which can directly affect the time transfer, data routing and hotspots problems. We assume that by placing the communicating tasks as close as possible, we can avoid long and costly communications which may increase the probability of having many hotspot points. This assumption is proved by the results in Fig 3.10. Compared with all of these mentioned methods, using VLS-BCF creates regularly less number of possible hotspots.

Comparison of VLS-BHF with MER-BottomLeft, MER-BestArea, MER-BestContact, MER-BestLong, MER-BestShort and VLS-FCF

As VLS-BHF privileges to reduce the number of hotspots point before to limit the communication cost, it is consistent that the number of hotspots using VLS-BHF is less than mentioned techniques which do not aim to minimize the hotspot points during their scheduling and placement. The number of hotspots produced by our method is reduced on average by 1,8% compared with the minimum number of hotspots produced by all the comparing methods and by 50,4% compared with the maximum number of hotspots. However, in some cases, for example cases 4 and 5 of Fig 3.10, VLS-BHF cannot guarantee the minimum hotspots due to the less number of point placement decisions compared with MERs methods.

In terms of communication cost, as the distance between communicating tasks is partially considered, VLS-BHF produces interesting solutions compared with mentioned techniques.

Comparison of VLS-BCF with VLS-BHF

In terms of communication cost, it is consistent that VLS-BCF achieves in average 10% reduction compared with VLS-BHF. On the contrary, in terms of number of hotspots, VLS-BHF allows on average 7,5% number of hotspots reduction compared with VLS-BCF. Depending on the targeted objective and the constrains of the application, we can decide to use VLS-BCF or VLS-BHF strategy.

3.3 Summary

In this chapter, we presented three different spatio-temporal scheduling strategies for two types of 2D homogeneous FPGA : the 2D Bloc Area FPGA and the 2D Free Area FPGA. The first algorithm called Pfair-ERR extends the famous Pfair algorithm to take into account the dependencies between tasks. Pfair-ERR aims at reducing the global communication cost between tasks. By favoring the direct communication with large amount of data and placing communicating tasks near each other, Pfair-ERR produces interesting solutions with a low global communication cost. However, these produced solutions do not take into consideration of the preemption and migration cost which are non negligible in the reconfigurable architecture. Therefore, using Pfair-ERR may not be the right choice to resolve the task scheduling and placement on the 2D Bloc Area.

The second algorithm called VLS-BCF and the third algorithm called VLS-BHF address the spatio-temporal scheduling of tasks for the 2D Free Area FPGA while not allowing the preemption and migration of the task. These algorithms are based on VLS technique which is a low complexity and a simple data structure technique. The objective of VLS-BCF is to minimize the direct communication cost between tasks by limiting the long communication with a large amount of exchanged data. VLS-BCF shows a significant reduction of communication cost compared to some existing methods. Moreover, we prove also that by applying the VLS-BCF, we can avoid long and costly communications which may increase the probability of having many hotspot points during the exchange of data. In order to evaluate more efficiently the VLS-BCF in terms of hotspot points, VLS-BHF is developed to compare with VLS-BCF. VLS-BHF privileges the reduction of the number of hotspot points before the limitation of communication cost. The results show that VLS-BHF is also an interesting technique while allowing a small number of hotspots reduction with a low communication cost overhead compared with VLS-BCF.

Chapter 4

Online spatio-temporal scheduling strategy for 2D heterogeneous reconfigurable resources

Compare to the 1D column and 2D homogenous Reconfigurable Resources (RR), the 2D heterogeneous RR imposes stricter placement constraints for tasks and requires a different strategy of placement. In this chapter, we present a new algorithm for online scheduling and placement of tasks on a 2D heterogeneous RR. Our method called Spatio-Temporal Scheduling for 2D Heterogeneous RR (STSH), aims at minimizing the overall execution time of an application executing on the 2D heterogenous RR.

The first section 4.1 presents the architecture model and the task model. Section 4.2 introduces the basic prefetching configuration scheduling and explains how is the importance of taking into account task priority and task placement decision during the scheduling. Section 4.3 formalizes the problem that our proposed algorithm addresses. Section 4.4 presents in details our algorithm. Section 4.5 shows some results obtained by applying our algorithm and section 4.6 concludes this chapter.

4.1 2D heterogeneous reconfigurable resources model and task model

Modern FPGAs embed not only CLBs but also BRAMs and DSPs that are partially reconfigurable as well. Placing a task on such 2D heterogeneous reconfigurable resources must respect the resources and the task placement constraints. Therefore, it is necessary to define a model for expressing the heterogeneity of a reconfigurable area and also of a task executing on this kind of platform. Details will be specified in the following of this section.

4.1.1 Platform description

Our target 2D heterogeneous reconfigurable resources is inspired from the embedded FPGA (eFPGA) layer of a 3D stacked chip in the on-going Flextiles project [79]. The 3D Flextiles chip, illustrated in Fig 4.1 , is composed of a manycore layer and a reconfigurable (eFPGA) layer. The manycore layer contains several General Purpose Processor (GPP) and Digital Signal Processor (DSP) cores. The reconfigurable layer, dedicated for HW accelerators, is used to provide a high level of flexibility by supporting the feature of dynamic reconfigurable paradigm. The NoC on the manycore layer is used to support the communication between processors, between processors and accelerators and between accelerators.

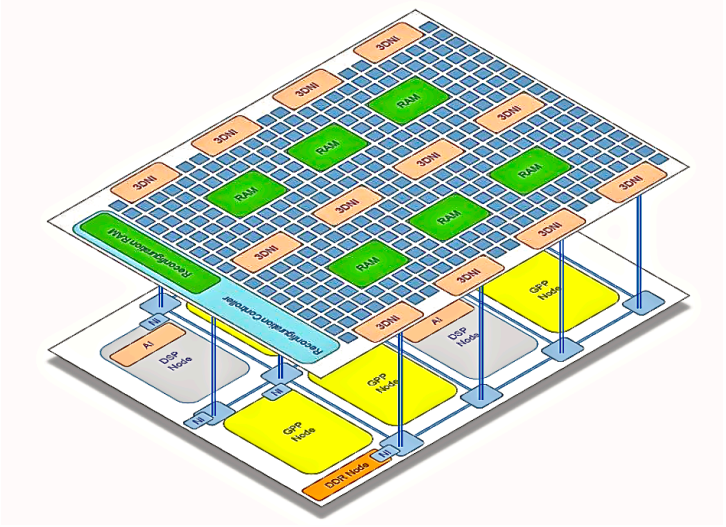


FIGURE 4.1: 3D Flextiles chip overview

In this chapter, we focus on the scheduling and placement of dependent tasks on a 2D heterogeneous reconfigurable architecture. Thus, we simplify the Flextiles architecture in order to reduce the complexity of the problem. However, this reduction does not affect the mean fullness of our approach but allows to focus on the essential research and evaluation. This architecture is based on a large reconfigurable area connected to one processor via a bus which supports the communication between accelerators, see Fig 4.2. The processor supports the execution of an Operating System able to control the task management. Furthermore, this architecture contains a shared memory for the storage of data transferred between tasks when needed.

The 2D heterogeneous FPGA has different types of resources, which are symmetrically located at fixed positions on the layer (see Fig 4.2): computing resources (configurable logic blocks - CLBs), memory resources (Blockram - BRAMs) for storing data during computations and Accelerator Interfaces (AIs) to provide access points between processors and accelerators. Fitting with the reconfigurable technology based on a virtual bitstream, the symmetry of resources on the layer allows a high flexibility in terms of task relocation and increases the designer degree of freedom for task placement.

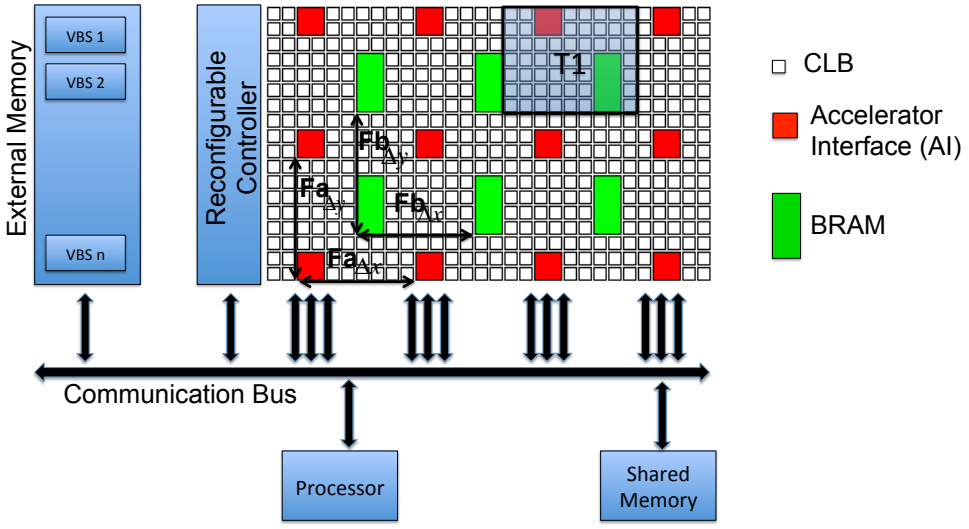


FIGURE 4.2: 2D Heterogenous FPGA

We consider that a task must contain at least one AI and one BRAM. The BRAM is used in each task to receive the data from other task and also to store data during its execution. The AI is used to homogenize the control of task by the processor, but also enables data transfer from/to SW and to/from HW execution. The functionality of the AI is supposed to be the same as in Flextiles and it offers various management services

as configuration, control, and debug. Each AI is connected to the communication bus as described in Fig 4.2. When an task covers more than one AI, one of them must become active and support the communications, others stay inactive. If two tasks need a data exchange, the operating system must configure two I/O AIs of these two tasks for setting up the communication. Then, data will be sent from one AI to another AI. The shared memory is needed when one task must send data to another task which has not been configured yet due to some dependencies.

4.1.2 2D heterogeneous model

The size of the RR is defined by W_{FPGA} * H_{FPGA}, with W_{FPGA} and H_{FPGA} the width and the height of the RR in number of logic elements. Due to the symmetry of the RR, the locations of BRAMs (or AIs) are defined by the distance between two adjacent BRAMs (or AIs) but also by the position of the first instance of the block. The horizontal (respectively vertical) distance between two BRAMs is $Fb_{\Delta x}$ (respectively $Fb_{\Delta y}$) and the position of the first BRAM instance is given by Fb_{x0} and Fb_{y0}. The same distances and initial position of first instance are defined for the AI, as $Fa_{\Delta x}$, $Fa_{\Delta y}$, Fa_{x0} and Fa_{y0}. The coordinates $(0, 0)$ corresponds to the bottom left corner of the RR.

From these values, the size of RR and the location of the different elements can be defined and all symmetrical FPGAs can be supported. A RR is then completely defined by

$$\begin{aligned} RR \quad = \quad & \{(W_{FPGA}, H_{FPGA}), (Fb_{x0}, Fb_{y0}), (Fb_{\Delta x}, Fb_{\Delta y}), \\ & (Fa_{x0}, Fa_{y0}), (Fa_{\Delta x}, Fa_{\Delta y})\} \end{aligned} \tag{4.1}$$

As an example, the RR represented in Fig 4.2 can be defined as

$$RR = \{(30, 18), (6, 3), (8, 8), (2, 0), (8, 8)\}$$

We call Nb_{total} the total number of BRAMs and Na_{total} the total number of AIs on the RR; Nb_{total} (respectively Na_{total}) is computed by an evaluation of the number of BRAMs (respectively AIs) which can be placed in the area defined by W_{FPGA} * H_{FPGA}.

This computation is given by

$$\begin{aligned} Nb_{total} &= (\lfloor (W_{FPGA} - Fb_{x0} - 1)/Fb_{\Delta x} \rfloor + 1) \\ &\quad *(\lfloor (H_{FPGA} - Fb_{y0} - 1)/Fb_{\Delta y} \rfloor + 1) \\ Na_{total} &= (\lfloor (W_{FPGA} - Fa_{x0} - 1)/Fa_{\Delta x} \rfloor + 1) \\ &\quad *(\lfloor (H_{FPGA} - Fa_{y0} - 1)/Fa_{\Delta y} \rfloor + 1) \end{aligned} \tag{4.2}$$

4.1.3 Task model

Reconfiguration of a HW task on such platform consists in allocating a rectangular zone and loading the bitstream towards the reconfiguration port. A task is supposed relocatable in several regions of the RR. A region on the RR is called the feasible region for a task when and only when all resources in this region are available and match those in the task. Fig 4.2 shows an example of a feasible region for T_1. This feasible region is a rectangular region whose the coordinates of the bottom left point is positioned relative to the point (0,0) of the RR.

In this context, as RR definition, the model of an HW task is defined by its size, and the location of each specific block included inside. So this model is defined as

$$M_i = \{(w_i, h_i), (Mb_{i,x0}, Mb_{i,y0}), (Ma_{i,x0}, Ma_{i,y0})\} \tag{4.3}$$

with w_i and h_i the width and height of the model M_i for a relocatable and reconfigurable task T_i, $(Mb_{i,x0}, Mb_{i,y0})$ the position of the first instance of BRAM in the model M_i, and $(Ma_{i,x0}, Ma_{i,y0})$ the first instance of AI in the model M_i. We define Nb_i as the number of BRAMs required by T_i, Na_i the number of AIs required by T_i. By applying the same calculation as Nb_{total} and Na_{total} for the task T_i, Nb_i and Na_i could be computed. As a task is a HW task, it is necessary to specify the reconfiguration time (HR_i) and the worst execution time (HE_i). The communication time between tasks is not considered since it is taken into account by the execution time. Thus, a task executing on a 2D heterogeneous RR is completely defined as

$$T_i = (HR_i, HE_i, M_i) \tag{4.4}$$

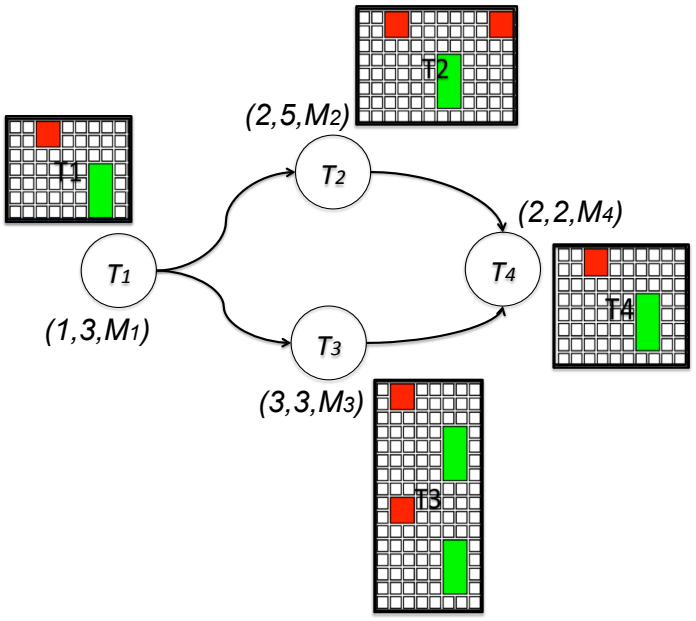


FIGURE 4.3: A set example comprising 4 tasks

We consider that an application is composed of a set of dependent tasks whose the execution flow is not known before the application gets starting. A task only has the knowledge about its successors during application execution when it is requested by the scheduler. When the task finishes its execution, it is removed from the RR and the resources occupied by this task can be used for another tasks. Fig 4.3 shows an example of a set comprising 4 tasks whose each task T_i is represented by (HR_i, HE_i, M_i). The parameter M_i of each task represents the corresponding model of the task T_i and is illustrated by the rectangle next to T_i. According to the model definition of task, we can describe T_1 = (1, 3, M_1) with M_1 = {(9, 7), (6, 0), (2, 5)} and T_2 = (2, 5, M_2) with M_2 = {(12, 8), (6, 1), (2, 6)}, etc.

4.2 Prefetching Configuration and Motivation

4.2.1 Prefetching for Partial Reconfigurable Architecture

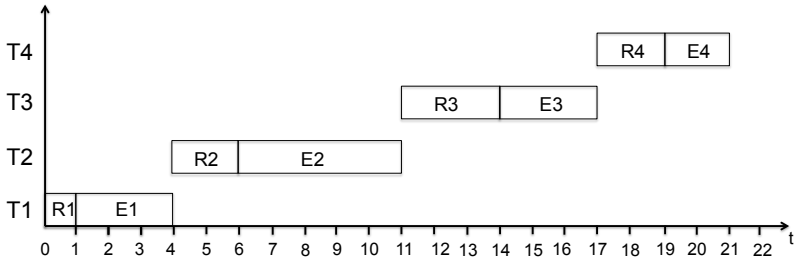
One of the main research interests of FPGA is dynamic and partial reconfiguration which allows a system to change a fraction of its resources at runtime without affecting the rest of the system. This feature obviously provides a higher flexibility and more powerful computing per area to deal with the dynamism of current multimedia applications requiring

a high performance. Nevertheless, one of the major overheads in reconfigurable computing is the time it takes to reconfigure tasks at run-time in the system. Depending on the size the task, DPR may create a costly reconfiguration overhead, therefore increases the application's runtime.

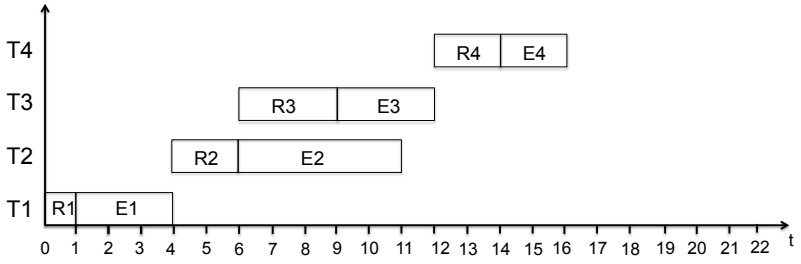
For example, reconfiguring a JPEG decoder task which occupies 30% of the Virtex XC2V6000 resources, requires 12 ms with the reconfiguration circuitry running at maximum speed [31] or reconfiguring a device with a partial bitstream of 1.1 Mbytes on a Virtex II-Pro could take 0.92 ms if the Internal Configuration Access Port (ICAP) is fed at the over-clock frequency of 300 Mhz [80]. Imagine if an application spends 40% of its time in reconfiguration and if we can reduce the overhead per reconfiguration by a factor of 2, we would at least reduce the application's runtime by 20%. Thus, developing techniques to minimize the reconfiguration time overhead, making the use of DPR more effective is crucial.

Among several techniques, configuration prefetching is known as an effective technique. This technique firstly introduced by S. Hauck [81] and then improved by Z. Li [82] allows to overlap the reconfiguration of a task during the execution of another task. In prefetching, a task can be loaded as soon as possible for starting its reconfiguration whenever the configuration controller is available and at least one feasible region is available for the task. Even if the task cannot execute immediately after its reconfiguration due the involvement of dependencies with other tasks, the fact of hiding the reconfiguration phase by loading the task during the execution of other tasks reduces significantly the reconfiguration overhead.

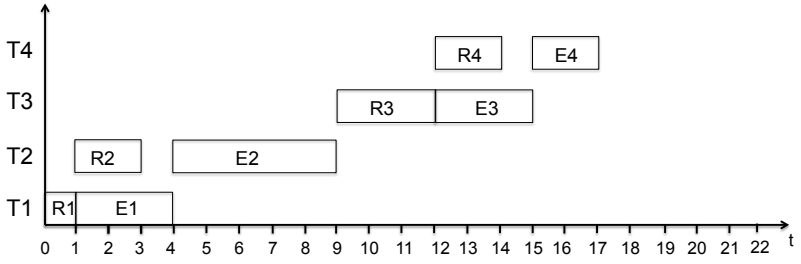
Different scheduling scenarios for the task set in Fig 4.3 are given in the Fig 4.4. Fig 4.4(a) shows the case where configuration prefetching is not at all applied. Only one task can be configured and executed at a time which leads to a high overall execution time. Fig 4.4(b) shows the case of lazy prefetching, i.e. the configuration prefetching is partially considered for only receiver tasks. Fig 4.4(e) shows the case where configuration prefetching is totally considered. The other cases will be discussed in the next section. We can see a big difference in terms of the overall execution time by applying different scheduling strategies.



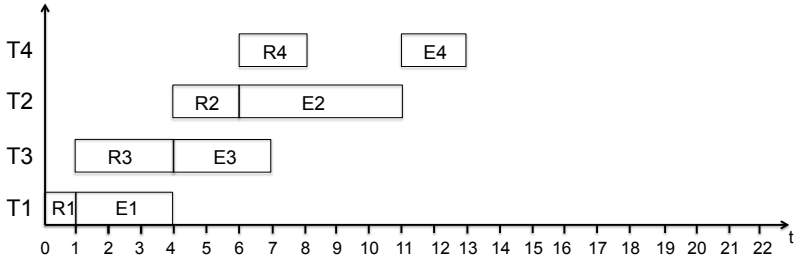
(a)



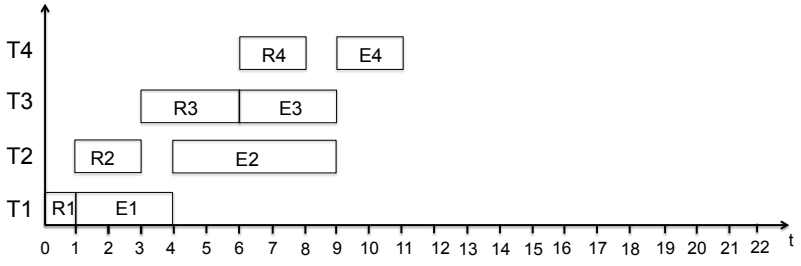
(b)



(c)



(d)



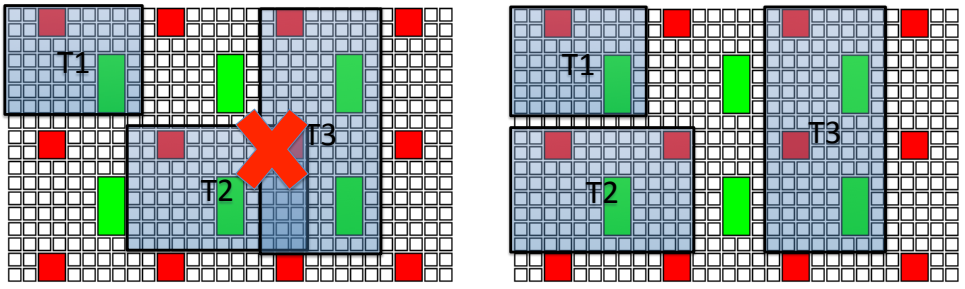
(e)

FIGURE 4.4: Different scenarios of task scheduling and placement for the task set in Fig 4.3

4.2.2 Task Priority

Despite the advantages offered by prefetching, this technique could be inefficient if it does not combine with the scheduler at run time to decide, among ready tasks, what task should be scheduled in what order. The ineffective order of tasks to schedule could increase the overall execution time. For example, according to the set of task in Fig 4.3, when T_1 finishes its reconfiguration, T_2 and T_3 both become ready to be reconfigured. At that time, the scheduler has the choice between requesting T_2 or requesting T_3 first. Fig 4.4(d) shows the case where T_3 is chosen first while Fig 4.4(e) shows the case where T_2 is chosen first. In these two cases, we supposed that when a task is requested, a feasible region for the task is available therefore the task can be immediately reconfigured.

4.2.3 Task Placement Decision



(a) (b)

FIGURE 4.5: -a- the placement of T_2 prevents the placement of T_3; -b- the placement of T_2 favors the placement of T_3

Also, due to an ineffective placement decision of a task, there is a possibility that this placement will prevent the placement of future tasks. Therefore, when next tasks are requested, they may not find feasible regions to be placed and must be delayed until feasible regions are available. Consequently, the execution time of the application will be penalized. Fig 4.4(c) shows the scheduling scenario where the placement of T_2 prevents the placement of T_3 (spatially represented as in Fig 4.5(a)) , thus T_3 must be delayed until T_2 finishes its execution and is removed from the FPGA. On the other hands, if T_2 is placed as in Fig 4.5(b), it will ease the placement of T_3 resulting a smaller overall execution time length. The scheduling scenario with the correct placement decision of T_2 is given in Fig 4.4(e).

4.2.4 Motivation

We have shown how the task priority and the placement decision are extremely important to reduce the reconfiguration overhead. Finding the optimal solution in terms of overall execution time is an NP-hard problem when task scheduling, placement and configuration prefetching need to be considered together. Moreover, in online scheduling, we cannot guarantee the best solution as the decisions are made on the fly. Different decisions of scheduling and placement could produce different solutions of overall execution time. Therefore, it is necessary to develop an online spatio-temporal scheduling strategy which is able to give an "approximate" solution in a "short" computing time. In this chapter, we propose a new strategy for run-time scheduling and placement of tasks on a 2D heterogeneous FPGA layer in order to minimize the overall execution time of the application. Our strategy tries to load tasks as early as possible while considering two factors: the priority of tasks to schedule and the placement decision to avoid conflicts between tasks.

4.3 Formalization of the spatio-temporal scheduling problem for 2D heterogeneous FPGA

The objective of our algorithm is to minimize the overall execution time of dependent tasks running on 2D heterogeneous FPGA. To achieve that, the scheduler must be able to handle all the following requests:

- what task to schedule at what time (temporal reconfiguration) ?
- where to place the task (spatial reconfiguration) ?
- when to start the execution of a task according with its precedence constraints (temporal scheduling) ?

An ILP formalization for scheduling and placement of tasks on 2D homogenous RR is described in [30]. As we consider the heterogeneity of the FPGA, different types of resources and also the placement constraints of tasks on these resources must be taken into account. In this part, we introduce a formalization with the main constraints of task scheduling and placement. We first define some variables as

- t_{hsr_i}: time that T_i is loaded in RR for starting its reconfiguration;
- t_{hse_i}: time that T_i starts its execution in RR;
- $Ah_{R_{k,i,t}} = 1$ if T_i is present at the region $R_{k,i}$ at time t, 0 otherwise. $R_{k,i}$ is defined below;
- $S_{R_{k,i,t}} = 1$ if there exists at least one feasible region $R_{k,i}$ for T_i at time t, 0 otherwise;
- $FR_t = 1$ if the reconfiguration port is occupied at time t, 0 otherwise. We suppose that only one reconfiguration port is possible, i.e. only one task can be loaded at a time;
- $Nb_{f,t}$: number of non-utilized BRAMs on RR at time t;
- $Na_{f,t}$: number of non-utilized AIs on RR at time t;
- t_{global}: the overall execution time of the task graph;

$$t_{global} = max\left\{(t_{hse_i} + HE_i)\right\}; \qquad \forall\, i = 1, \ldots, N_T \tag{4.5}$$

with N_T the total number of tasks in the graph

4.3.1 Objective

Minimizing the overall execution time of the task graph defined as

$$min\left(t_{global}\right) \tag{4.6}$$

This minimization must respect all the constraints that we present below.

4.3.2 Resource Constraints

At least a feasible region $R_{k,i}$ is found to accommodate the task T_i at time t is presented as:

$$\exists\, R_{k,i} \mid S_{R_{k,i,t}} = 1 \qquad \forall i = 1, \ldots N_T \tag{4.7}$$

$R_{k,i}$ is a rectangular region defined as

$$R_{k,i} = \{Rx_k, Ry_k, Rx_k + Mw_i, Ry_i + Mh_i\} \tag{4.8}$$

with Rx_k and Ry_k the coordinates of $R_{k,i}$, Mw_i and Mh_i the width and the height of the task T_i but also of $R_{k,i}$. We call $R_{k,i}$ a feasible region for T_i at time t when and only when it satisfies three following conditions:

- The number of BRAMs (respectively AIs) required by T_i must be less than the number of non-utilized BRAMs (respectively AIs) on the FPGA at time t

$$Nb_i < Nb_{f,t} \quad \wedge \quad Na_i < Na_{f,t} \tag{4.9}$$

  $Nb_{f,t}$ and $Na_{f,t}$ are calculated by:

$$\begin{aligned} Nb_{f,t} &= Nb_{total} - \sum_j Nb_j * Ah_{R_{l,j,t}} = 1 \\ Na_{f,t} &= Na_{total} - \sum_j Na_j * Ah_{R_{l,j,t}} = 1 \end{aligned} \tag{4.10}$$

- If T_i is placed at coordinate (Rx_k, Ry_k), all resources of T_i must fit on available resources of RR. This condition can be expressed by the following constraints

$$\begin{aligned} Rx_k + Mb_{i,x0} &= Fb_{x0} + m_1 * Fb_{\Delta x} \mid \exists\, m_1 \in N \\ Ry_k + Mb_{i,y0} &= Fb_{y0} + m_2 * Fb_{\Delta y} \mid \exists\, m_2 \in N \end{aligned} \tag{4.11}$$

$$\begin{aligned} Rx_k + Ma_{i,x0} &= Fa_{x0} + m_3 * Fa_{\Delta x} \mid \exists\, m_3 \in N \\ Ry_k + Ma_{i,y0} &= Fa_{y0} + m_4 * Fa_{\Delta y} \mid \exists\, m_4 \in N \end{aligned} \tag{4.12}$$

  If T_i is placed at coordinate (Rx_k, Ry_k) of RR, the first BRAM instant of T_i must be found at the relative position $(Mb_{i,x0}, Mb_{i,y0})$, or absolute position $(Rx_k + Mb_{i,x0}, Ry_k + Mb_{i,y0})$. The equation 4.11 verifies whether the first BRAM instant of T_i matches one of the BRAMs of the RR. We use the symmetric formalization of the RR platform to verify if a BRAM of RR is located at the position $(Fb_{x0} + m_1 * Fb_{\Delta x}, Fb_{y0} + m_2 * Fb_{\Delta y})$. If $m_1, m_2 \in N$ exist, it means that a BRAM is present at this location. The same verifications are done in the equation 4.12 for the first AI instant of T_i.

- The rectangle $R_{k,i} = (Rx_k, Ry_k, Rx_k + Mw_i, Ry_k + Mh_i)$ does not overlap with other tasks currently placed on the RR and does not be larger than RR size.

$$\begin{aligned} \nexists\, R_{l,j} \quad | \quad & Ah_{R_{l,j,t}} = 1 \; \forall j \neq i \\ \wedge \quad & Rx_k < (Rx_l + Mw_j) \\ \wedge \quad & (Rx_k + Mw_i) > Rx_l \\ \wedge \quad & Ry_k < (Ry_l + Mh_j) \\ \wedge \quad & (Ry_k + Mh_i) > Ry_l \\ \wedge \quad & (0 \leq Rx_k < Rx_k + Mw_i \leq W_{FPGA}) \\ \wedge \quad & (0 \leq Ry_k < Ry_k + Mh_i \leq H_{FPGA}) \end{aligned} \tag{4.13}$$

  The first five lines of the constraint defined in equation 4.13 verifie that it does not exist another rectangle $R_{l,j} = (Rx_l, Ry_l, Rx_l + Mw_j, Ry_l + Mh_j)$ currently on the RR for which there is an intersection with $R_{k,i}$. The last two lines of the constraint mean $R_{k,i}$ must be located inside the RR area.

4.3.3 Reconfiguration Constraints

A task T_i is ready to be scheduled at time t when and only when all the predecessors of T_i finished their reconfigurations, i.e:

$$t \geq max(t_{hsr_j} + HR_j) \qquad \forall j \mid T_j \in Pred(T_i) \tag{4.14}$$

A task T_i is said reconfigurable at time t when and only when it satisfies the following conditions:

- T_i is ready to be scheduled, i.e. the equation 4.14 is satisfied
- The configuration port at time t is free, i.e:

$$FR_t = 0 \tag{4.15}$$

- At least one feasible position $R_{k,i}$ for T_i is found, i.e:

$$\exists\, R_{k,i} \mid S_{R_{k,i},t} = 1 \tag{4.16}$$

$t_{hsr_i} = t$ when the scheduler decides to reconfigure T_i at this time.

4.3.4 Execution Constraints

Due to the data dependencies, a task T_i is ready to be executed at time t when and only when it satisfies the following conditions:

- all predecessors of T_i finished their executions, i.e:

$$t \geq max(t_{hse_j} + HE_j) \qquad \forall j \mid T_j \in Pred(T_i) \tag{4.17}$$

- the reconfiguration of T_i is finished, i.e:

$$t \geq t_{hsr_i} + HR_i \tag{4.18}$$

When a task T_i is ready to be executed, it will be executed immediately without the scheduler decision, therefore $t_{hse_i} = t$

4.4 Spatio-Temporal Scheduling for 2D Heterogeneous Reconfigurable Resources (STSH)

The algorithm consists in prefetching tasks as soon as possible while considering two factors: the priority of new tasks to scheduled and the placement decision to avoid conflicts between tasks. The first part of this section presents the pseudocode of our proposed strategy STSH, the second part details the task placement and an example is given at the end of this section to ease the understanding of our STSH.

4.4.1 STSH Pseudocode

Algorithm 1 shows the pseudocode of our strategy. It is related to the spatio-temporal scheduling by determining when to reconfigure what task and where. We list here the parameters used in the pseudocode:

- t: current tick time of the operating system. t is related to the time when a task starts (or finishes) its reconfiguration or its execution;
- $timeList$: list of the tick time t for the operating system;
- $currentTasks$: list of tasks present on RR at time t ;
- TL: scheduling list containing the tasks ready to be scheduled at time t;
- L: list of next available successors of T_i whose predecessors (except T_i) have all been scheduled and allocated;
- PL: list containing tasks ready to be scheduled at time t and next available successors of T_i. PL is the fusion list of TL (without T_i) and L;
- RR, FR, HR_i, HE_i, t_{hse_i}, $R_{k,i}$: parameters defined in the previous section;

These parameters will be updated at each time t.

The *Algorithm* 1 works as follows. At the beginning, t equals to 0, $timeList$ is empty and TL contains only the source tasks, i.e. the tasks which do not have any predecessors (line [2-4]). The main part of the algorithm is presented from line [5-25] detailing what happens at each tick time t. It is repeated until TL is empty, i.e. when all the tasks have been scheduled.

The function *Update* checks whether a task finishes its reconfiguration or its execution at current time t and then updates the RR state, the $currentTasks$ list and the reconfigurator port FR state. FR is busy during the reconfiguration phrase of a task and becomes free when a task finishes its reconfiguration. When a task finishes its execution, it will be removed from the RR and also from the $currentTasks$ list.

After the updating phase, the scheduler will schedule and place tasks of TL one by one. The order of tasks to schedule follows their order in TL. At time t when T_i is

Algorithm 1 STSH strategy

```
1: function STSH {{Tasks}, RR}
2: t = 0
3: timeList = ∅
4: TL = {T_i}        | Pred(T_i) = ∅        ∀ T_i ∈ {Tasks}
5: while TL ≠ ∅ do
6:    Update(RR, currentTasks, FR)
7:    for all T_i ∈ TL do
8:       Establish L: {T_j = NextAvailableSuccessors(T_i) | T_j ∉ TL}
9:       PL = TL\ T_i ∪ L
10:      PL.SortExecutionTime()
11:      R_{k,i} ← FindFewestConflictRegion(T_i, PL)
12:      if ∃R_{k,i} and (FR_t == 0) then
13:         FR_t = 1
14:         RR.Load(T_i)
15:         currentTasks.Add(T_i)
16:         timeList.Add(t + HR_i, t_{hse_i}, t_{hse_i} + HE_i)
17:         TL.Delete(T_i)
18:         TL.Add(L)
19:         TL.SortExecutionTime()
20:         break;
21:      end if
22:   end for
23:   timeList.delete(t)
24:   t = min(timeList)
25: end while
26: }
```

requested, the algorithm tries to find the feasible region $R_{k,i}$ for placing T_i on the RR in order to favor the placement of the next reconfigurable task. This region $R_{k,i}$ found by $findFewestConflictRegion(Ti, PL)$ function, must satisfy the conditions for the resource constraints mentioned in the formalization part. The details about this function will be specified in the next. If $R_{k,i}$ for T_i exists (condition 4.16) and the reconfigurator port FR is free (condition 4.15), T_i will be loaded into the RR for starting its reconfiguration. Consequently, $timeList$ and $currentTasks$ will be updated, available successors of T_i will also be added to TL. Then, to reduce the reconfiguration overhead, tasks in TL will be sorted in descending order of their execution time.

Finally, we jump to the next tick time by performing the minimal value in $timeList$ (line [23-24]) and continue to schedule next task in TL until TL is empty.

4.4.2 Placement

This part details the technique used in the function $findFewestConflictRegion(T_i, PL)$ called by the algorithm 1. The technique aims to find the most suitable position for T_i at time t when it is requested. This technique is divided into two parts: the first part called *Fast Feasible Region Search* serves to find quickly all feasible regions for the task T_i at time t on the RR, the second part called *Avoiding Conflict Technique* allows to evaluate all the feasible regions for T_i, then choose the most suitable one to place T_i.

4.4.2.1 Fast Feasible Region Search

To the best of our knowledge, most of the HW task placement algorithms deal with 1D or 2D FPGA homogenous architecture, for example KAMER [39], Vertex List [45], etc. However, real FPGAs have BRAM blocks, multipliers and DSPs in a certain disposition and this heterogeneity imposes stricter placement constraints for the task. Few algorithms deal with task placement on 2D heterogeneous architecture. M.Koester [47] proposed a placement algorithm which is able to deal with the constraints of the HW tasks. However, feasible positions of the task are not found at run-time. For each HW task, the given set of feasible positions is predefined at design time. Eiche et al. [48] implemented an on-line placer for heterogeneous devices by using a discrete hopfield neuronal network. They consider that the RR is divided in several Partial Regions and a task must contain at least one of them. This consideration can create a waste of resources when a task does not need the entire resources in the PRR.

In this part, we propose an efficient method allowing to quickly find all reconfigurable regions $R_{k,i}$ on the 2D heterogeneous RR where T_i can be allocated at time t. We define FP as the set of these feasible regions ($FP = \{R_{k,i}\}$). Instead of seeking each logic block along W_{FPGA} and H_{FPGA} of the RR to find a feasible region $R_{k,i}$, which is very time consuming, we propose to directly seek by the resources having fewest number on the RR. In our case, the number of BRAMs is less compared to AIs and CLBs, therefore we start scanning with BRAMs. The idea is to try to place the task in the region where the first BRAM instance of T_i matches with a BRAM of the RR and checking whether the region created is a feasible region for the task T_i. Thus, by placing the first BRAM instance of T_i successively on all BRAMs of the RR, all feasible regions could be found. By doing this

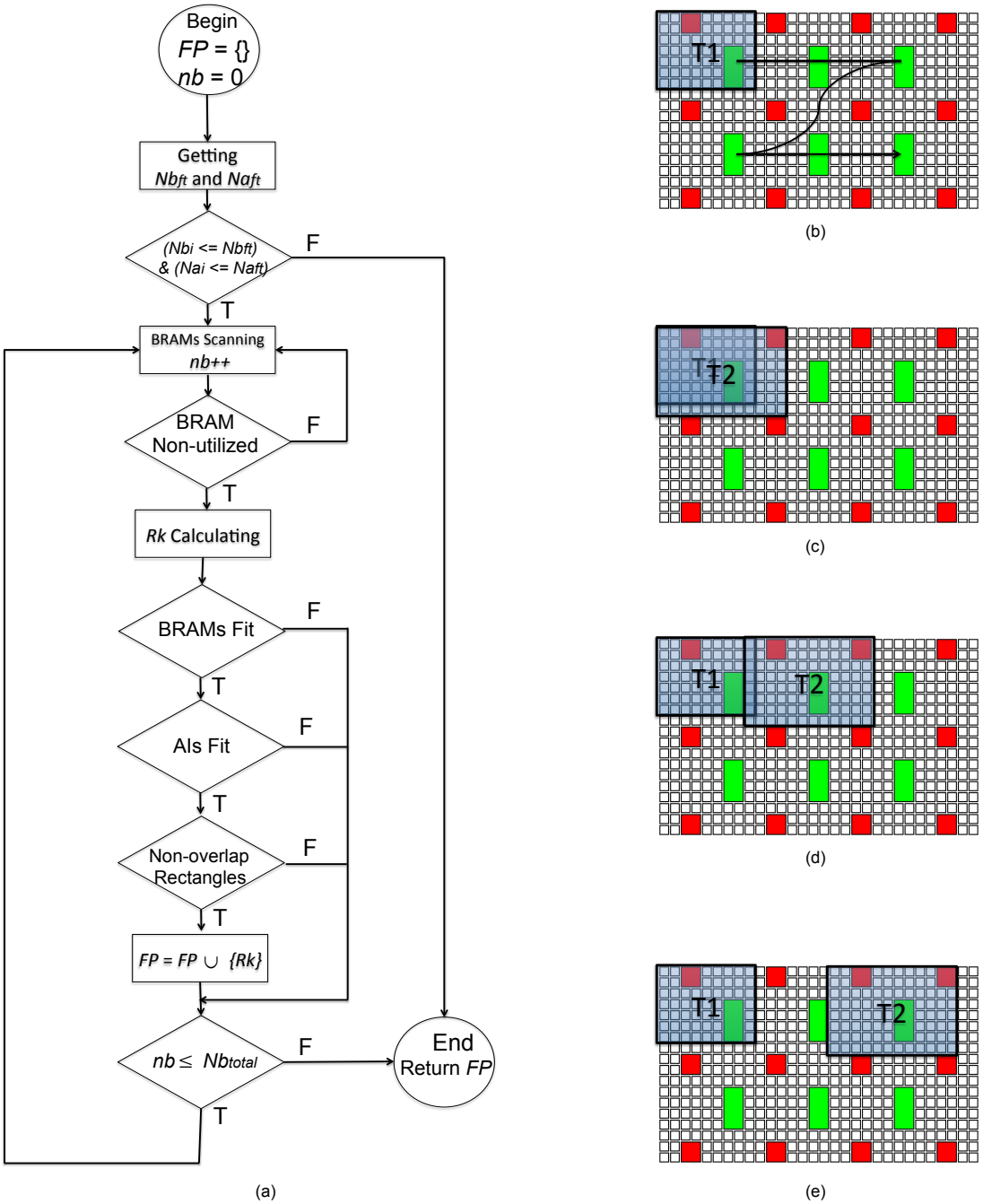


FIGURE 4.6: Quick search method for finding all feasible regions $R_{k,i}$ for the task T_i at time t

way, the complexity for searching feasible regions is reduced $(W_{FPGA}{}^{*}H_{FPGA})/Nb_{total}$ times.

Fig 4.6(a) represents the proposed searching method. Before scanning BRAMs, we realize a first check about the number of non-utilized BRAMs ($Nb_{f,t}$) and AIs ($Na_{f,t}$) on the RR at time t (condition 4.9). If the number of BRAMs and AIs required by the task are superior to the number of non-utilized BRAMs and AIs on the RR, no feasible region is available for T_i. On the contrary, we could start BRAMs scanning process. The BRAMs scanning step consists in scanning each BRAM of the RR one by one from the left to the right and then from the top to the bottom as in the figure 4.6(c). For each scanning iteration nb, if the BRAM is utilized (which is verified by BRAM non-utilized condition), we go to the next iteration. If not, we calculate the region $R_{k,i}$ where the first BRAM instance of T_i can be matched with the BRAM of the RR. $R_{k,i}$ is a valid feasible region for the task T_i when the conditions 4.11 are satisfied. BRAMs Fit Condition is used to check if all other BRAMs of the tasks are matched on the non-utilized BRAMs. If it is the case, the same process is done for AIs (condition 4.12). When BRAMs Fit condition and AIs Fit condition are satisfied, Non-overlap Rectangles (condition 4.13) must be respected to ensure that the created region $R_{k,i}$ will not overlap with other regions where other tasks are running and/or with the border of the RR. Once $R_{k,i}$ is a valid feasible region, it is added to FP and we scan the next BRAM until all BRAMs of the RR are scanned.

Fig 4.6(b) presents an example with T_1 is currently on the RR and we must find all feasible regions for T_2. Fig 4.6(c) shows an invalid region for T_2 with BRAM Non-utilized condition not satisfied, Fig 4.6(d) shows an invalid region with BRAMs Fit and AIs Fit condition satisfied but Non-overlap rectangles condition not satisfied. Fig 4.6(e) shows a valid feasible region for T_2.

4.4.2.2 Avoiding conflicts technique

At time t, once all feasible regions for T_i are found, $findFewestConflictRegion(T_i, PL)$ chooses the most suitable one that will create fewest conflicts with the next reconfigurable task to place T_i. As when a task T_i is requested, the scheduler has also the information about its successors. Thus, the placement decision of T_i has to take into account not only the ready tasks but also its next available successors. We call PL the list containing the ready tasks and the next available successors of T_i (line 9). The next reconfigurable task

is the first task in PL that may be loaded (may be not the first requested) into the RR after T_i. In order to ensure that the task having the most costly execution time will be requested at the next tick time, PL is sorted in the descending order of the execution time (line 10).

Firstly, the set of feasible regions for T_i, noted FP is found by using *Fast Feasible Region Search* presented in the previous part. Then, the next reconfigurable task is the first task T_j in PL satisfying the following condition: among all feasible regions of T_i, there must exist at least one feasible region $R_{k,i}$ so that if T_i is placed at this region, the RR will still have enough space to accommodate T_j. In the case a next reconfigurable task exists, our placement decision of T_i is based on the region $R_{k,i}$ giving the largest number of future feasible regions of T_j. If there are many $R_{k,i}$ which give the same largest number of future feasible regions of T_j, T_i will be placed at the furthest region from the RR center. Also, in the case there is no next reconfigurable task, T_i will be placed at the furthest region from the RR center.

	$R_{1,i}$	$R_{2,i}$	$R_{3,i}$	$R_{4,i}$
PL[0]	0	0	0	0
PL[1]	0	0	0	0
PL[2]	1	3	2	1
PL[3]	-	-	-	-

TABLE 4.1: T_i will be placed at $R_{2,i}$ in order to favor the placement of the next reconfigurable task PL[2]

Tab 4.1 shows an example that we have to choose the best feasible region among $\{R_{1,i}$, $R_{2,i}$, $R_{3,i}$, $R_{4,i}\}$ to favor the next reconfigurable task in PL. The list PL is sorted in descending order of the execution time, thus PL[0] will be requested first, then PL[1], PL[2] and PL[3]. The number inside the table means the total number of future feasible regions for the task in PL if T_i is placed at different $R_{k,i}$. According to the table, PL[2] is the next reconfigurable task and $R_{2,i}$ is chosen to place T_i. Once the next reconfigurable task is found, there is not anymore necessary to evaluate remaining tasks in PL.

4.4.3 Example

Fig 4.7 shows by time steps the scheduling and placement of task set defined in Fig 4.3. Three different colors are used to differentiate the reconfiguration phase (brown), the pending for the execution phase (gray) or the execution phase of a task (yellow).

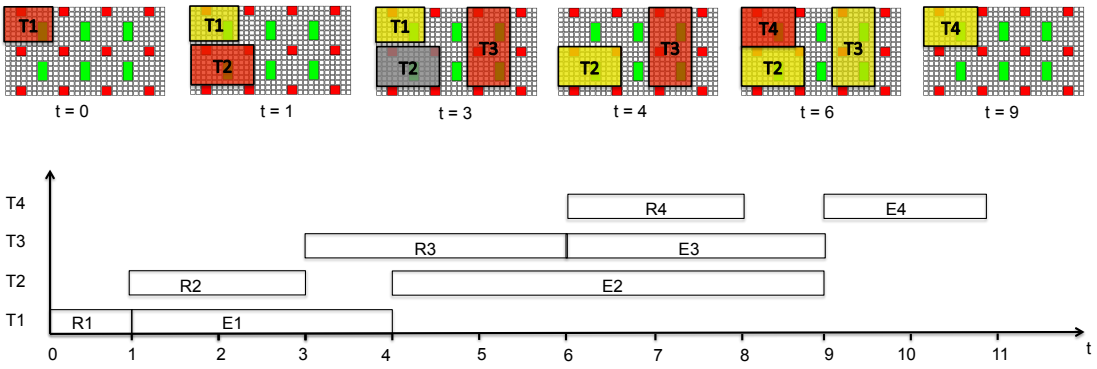


FIGURE 4.7: Scheduling and placement of tasks on 2D heterogenous FPGA

At time 0

- At the beginning, the scheduling list TL contains only T_1. TL=$\{T_1\}$
- Next available successors of T_1 are T_2 and T_3, thus T_1 is placed as in Fig 4.7 at time 0 to favor the placement of the next reconfigurable task T_2
- Once T_1 is reconfigured, it is removed from TL and T_2, T_3 are added to TL. TL=$\{T_2, T_3\}$

At time 1

- T_1 finished its reconfiguration and starts its execution. At that time, the reconfigurator becomes free and T_2 can start the reconfiguration.
- The placement of T_2 must favor the placement of next reconfigurable task which is T_3, therefore T_2 is placed as in the figure.
- Once T_2 is reconfigured, it is removed from TL. TL=$\{T_3\}$

At time 3

- T_2 finishes its reconfiguration but it cannot start the execution phase due to the unfinished execution of T_1.
- The reconfigurator is free at time 3, thus T_3 can start its reconfiguration. The placement of T_3 must favor its successor T_4. However, only feasible region is available for T_3, thus it is placed as in the figure.
- Once T_3 is reconfigured, it is removed from this list and T_4 is added to TL. TL=$\{T_4\}$

At time 6

- T_3 finishes its reconfiguration and starts its execution.
- T_4 can also be reconfigured on the top left of the RR
- Once T_4 is reconfigured, it is removed from this list. TL=$\{\emptyset\}$, i.e. T_4 is the last task to be scheduled.

At time 8

- T_4 finishes its reconfiguration but cannot start its execution due to the dependencies with T_2 and T_3.

At time 9

- T_2 and T_3 finishes their executions and are removed from the RR.
- T_4 can now start executing

At time 11

- T_4 finishes its execution and is removed from the RR.
- As T_4 is the last task of the task set, we mark 11 as the overall execution time for the task set.

4.5 Evaluation

In order to evaluate the quality of our proposed method, we generate different task sets and compare the results produced by our method to others. To our best knowledge, no standard benchmarks are available to evaluate online spatio-temporal scheduling algorithms. Therefore, we use our own synthetic benchmark sets.

First evaluation: Comparison with FFLP, RFWP, Napoleon-ex

For the first evaluation, we compare our proposed STSH with three different methods: i) First Fit using Lazy Prefetching (FFLP), ii) Random Fit With Prefetching (RFWP) and iii) Napoleon (NapoleonEx). The FFLP algorithm does not consider the prefetching technique and the order of tasks to schedule. In FFLP, tasks are placed at the first available region. RRWP considers the prefetching technique but not the order of tasks. In

RRWP, tasks are placed randomly on the FPGA at one of its feasible regions. Napoleon's heuristic [30] is very close to our strategy by considering the prefetching technique and also the task placement. Compared to the off-line Napoleon, our strategy has two different points: i) While Napoleon algorithm computes the order of tasks to schedule at design-time, our algorithm attributes the priority for the tasks at run-time. ii) The placement in Napoleon algorithm addresses the 2D homogenous FPGA and it uses furthest placement criteria in order to increase the probability of placing quickly large modules. In our method, the placement takes care of 2D heterogeneous FPGA and the new task will be placed in the region creating fewest conflicts with other tasks. To compare our algorithm with Napoleon, we extend the basic Napoleon to take into account of the heterogeneity of the task and the FPGA.

We generate 10 task sets (TS) whose the number of tasks N_T ranges from 5 to 14 tasks for each set. The task characteristic for each TS is shown in Tab 4.2. We note $PD[a-b]$ as the parallelism degree in the set, i.e except the sink task, a task has at least a successor and at most b successors. The RR used for the first evaluation is described as RR = {(36,34), (6,3), (8,8), (2,0), (8,8)}. For this RR model, a task with the characteristics in Tab 4.2 can occupy from 3% to 26,5% of the total RR's resources.

PD	HR_i	HE_i	w_i	h_i	M_i
[1-3]	[1-10] time units	[1-10] time units	[6-18] CLB sizes	[6-18] CLB sizes	[3%-26,5%] of RR resources

TABLE 4.2: Task set characteristic

TS	N_T	**FFLP**	**RFWP**	**NapoleonEx**	**Proposed STSH**
TS1	5	41	39.06	39	33
TS2	6	48.06	39	39	33
TS3	7	60.43	54	54	53
TS4	8	56.5	41.63	41	40
TS5	9	62.88	48	48	44
TS6	10	68.41	52.18	51	51
TS7	11	79.84	68	68	66
TS8	12	69.88	56	56	57
TS9	13	74.98	58.48	58	58
TS10	14	102.7	87	87	84

TABLE 4.3: Comparisons of the overall execution time for different techniques

The table 4.3 shows the comparisons of the overall execution time produced by our proposed STSH with FFLP, RFWP and Napoleon extension. In this table, 10 task sets are analyzed. Because FFLP, RFWP and Napoleon do not consider the order of tasks to

schedule, thus at time t when several tasks are ready to be scheduled, a task is chosen randomly among these tasks to be scheduled. In our examples, for each set, FFLP, RFWP and Napoleon are run 100 times each with the order of tasks is randomly chosen every time. The values produced by FFLP, RFWP and Napoleon in the table 4.3 are the average overall time of these 100 times. By performing the random order of tasks, we can compare fairly our method with others.

For almost cases, our STSH gives the shortest overall execution time. According to the table 4.3, STSH considering the order of tasks to schedule and the avoid conflict placement technique significantly reduces by 22,5% the overall execution time compared to FFLP and approximately 5% compared to RFWP and Napoleon. In the $TS8$ with N_T equals 12, the overall execution time produced by STSH is slightly greater than RFWP and Napoleon. This difference is due to the fact that the scheduler does not have the entire information about the execution flow of the set. The information about the successors of a task is only known when the task is requested to schedule, therefore at the time t the scheduling list contains only the tasks which will be ready to be scheduled in the near future from t. Then, our proposed STSH favors the placement of the next reconfigurable task but cannot predict the placement of far future tasks. However, in some cases, the placement of the task T_i at the time t can impact on the placement of T_j that was not in the scheduling list at that time.

Second evaluation: Comparison of different methods in the case the same task sets are run on the architectures having different sizes

We evaluate the efficiency of our proposed STSH by comparing with First First using Prefetching (FFP) and NapoleonEx. For these comparing methods, two cases are analyzed: the case the task priority i.e. the order of tasks to schedule is not considered (noted NoSort or NS) and the case the task priority is considered (noted Sort). We also evaluate how our method reacts when the task priority is not considered (noted STSH NoSort).

For that, we generate 100 different task sets with the characteristics given in Tab 4.2. These task sets are run on two different RRs, the first is described as RR_1 = {(36,34), (6,3), (8,8), (2,0), (8,8)} and the second is RR_2 = {(28,26), (6,3), (8,8), (2,0), (8,8)}. These two RRs are chosen in order to analyze the efficiency of our method compare with others in the case of a big architecture model as RR_1 and in the case of a small

architecture model as RR_2. As the RR_1 is bigger than RR_2, whatever the using methods, there will be more chance for requested tasks to be placed immediately without waiting until the RR is free. Thus, the task prioritiy and avoiding conflicts technique used in our method may not make big differences compared with other methods. Executing the same task sets within a smaller RR size as RR_2 is more challenging. When a task is requested, due to the limited resources of the RR, a wrong placement decision for the task may prevent the placement of next ready tasks, thus has a severe penalty on the overall execution for the task set. In this situation, our method may show the advantages compared with other methods.

Fig 4.8 shows the comparison of our method with others for 100 task sets executing on the RR_1. Fig 4.9 shows the comparison of our method with others for these 100 task sets on the RR_2. np_sup represents the number of cases that our method performs better than the other in terms of overall execution time, np_equal represents the number of cases that our method performs the same as the other method in terms of overall execution time, np_less represents the number of cases that our method performs less than the other in terms of overall execution time.

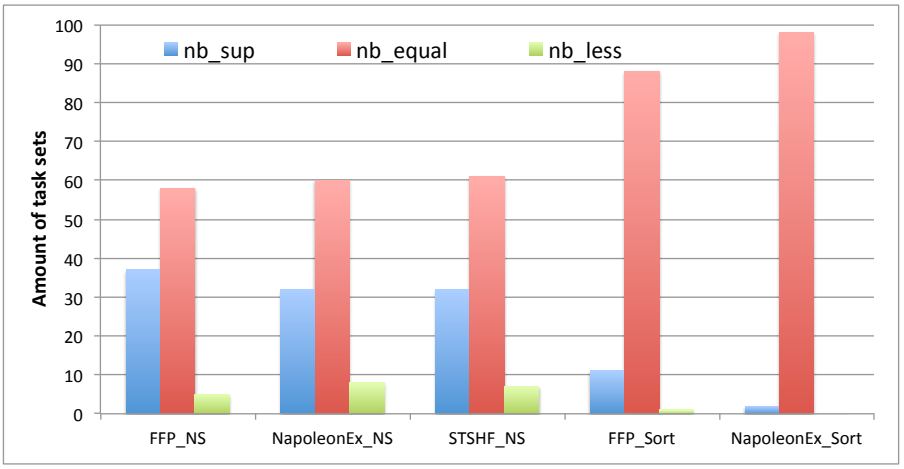


FIGURE 4.8: Comparison of our method with others for 100 task sets executing on RR_1 = {(36,34), (6,3), (8,8), (2,0), (8,8)}

For RR_1, Fig 4.8 proves that even the number of cases that our method performs better than NoSorting methods (FFP_NS, NapoleonEx_NS, STSH_NS) is about 30 to 40 cases among 100 cases, the number of cases that our method performs the same as other methods, are still higher. For other sorting methods (FFP_Sort and NapoleonEx_Sort), our methods performs, in most cases, the same as others.

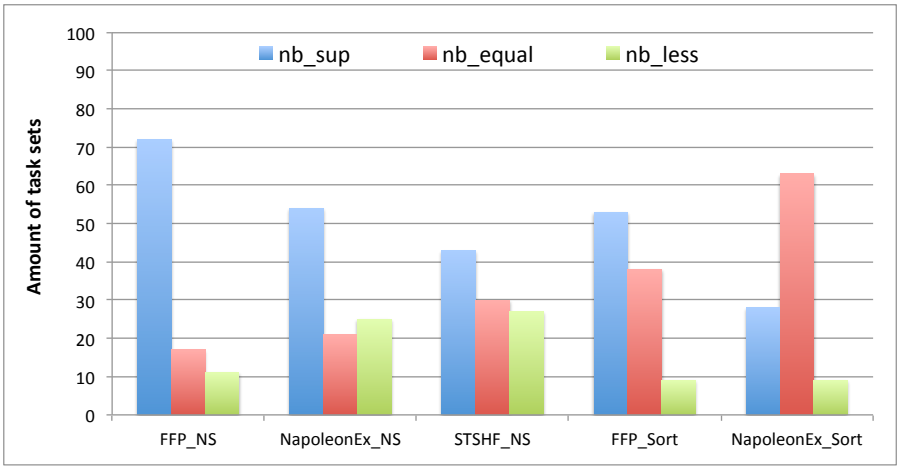


FIGURE 4.9: Comparison of our method with others for 100 task sets executing on RR_2 = {(28,26), (6,3), (8,8), (2,0), (8,8)}

For RR_2, in most cases, our method performs better than others thanks to the task priority and avoiding conflicts technique. Fig 4.9 shows a bigger nb_sup and less nb_equal than in the RR_1 architecture for each comparing method. As explained earlier, in few cases, our method performs less than others due to the unknown execution flow of the task set, thus our method cannot guarantee the optimal solution. However, the nb_less stays quite low.

Third evaluation: Comparison in terms of "exploited computation resources"
Even the main objective of our method is to reduce the overall execution time, we assume that this objective can also lead to a better exploited computation resources. For a set of dependent tasks, the exploited computation resources ECR are calculated as

$$ECR = \sum_{i=1}^{N_T} \frac{(t_{hse_i} + HE_i - t_{hsr_i}) * w_i * h_i}{t_{global} * W_{FPGA} * H_{FPGA}}$$

Fig 4.10 shows the comparison of our method with others in terms of average exploited computation resources for 100 task sets. As explained earlier, the difference between our algorithm and others while executing a task set in a big RR is not really large. Executing the same task set on a smaller RR as RR_2 can prove the advantages from our proposed method. Fig 4.10 confirms our explanation by showing that the ECR gain obtained for the RR_1 is also smaller than for the RR_2.

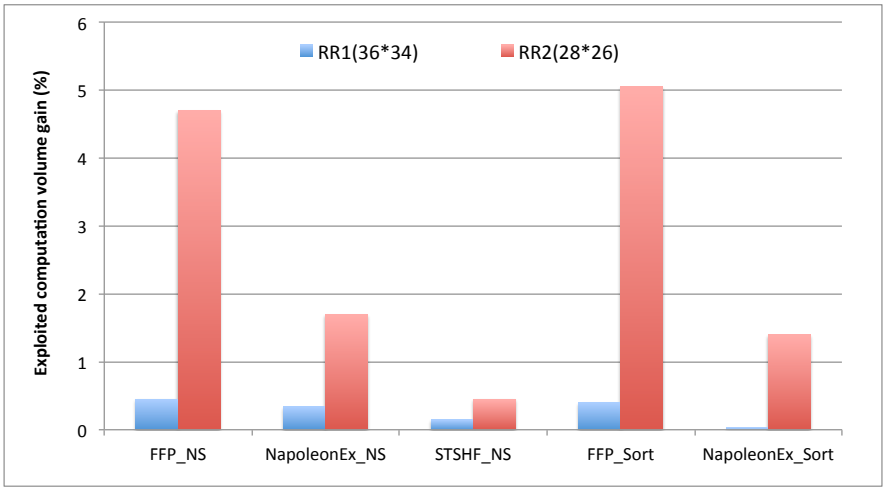


FIGURE 4.10: Comparison of our method with others in terms of exploited computation resources

4.6 Conclusion

In this chapter, we introduce first the formalization of the spatio-temporal scheduling problem for 2D heterogeneous FPGA. We present as well a technique allowing to quickly search feasible region of the task on this 2D architecture. Finally, we propose the Spatio-Temporal Scheduling for 2D Heterogeneous Reconfigurable Resources (STSH) method allowing to minimize the overall execution time of a task set executing on this 2D architecture. Our proposed strategy integrates prefetching technique while considering two factors: the priority of new tasks to schedule and the placement decision to avoid conflicts between tasks. The results show that our proposed strategy significantly reduces the overall execution time compared to some non-prefetching and other prefetching methods. It also leads to a better resource utilization compared to others.

For the next chapter, we will extend this work to take into account also the software executions, thus a task can exist in hardware and/or software. An efficient scheduler will be proposed in order to find a "near optimal" solution in term of the overall time of the application.

Chapter 5

Online spatio-temporel scheduling strategy for 3D Reconfigurable System On Chip

In previous chapters, we studied online spatio-temporal scheduling strategies for different types of FPGA. Our previous strategies exploited the dynamic partial capacity of the FPGA and tried to use Reconfigurable Resources (RR) as efficiently as possible. In such system as 3DRSoC, the use of FPGA is extremely important to achieve the high performance. Therefore, even the studies on the FPGA are not the main approaches of this thesis, they need to be analyzed before we go deep in the 3DRSoC world.

In this chapter, we extend the proposed strategies developed in chapter 3 and chapter 4 in order to take into account the 3rd dimension in 3DRSoC. For that, not only the execution of hardware (HW) tasks on the FPGA but also the one of software (SW) tasks on the processors needs to be managed in an efficient way to exploit at maximum the advantages offered by 3DRSoC systems. To address this topic, this chapter presents online spatio-temporal scheduling strategies for two 3DRSoC architectures: the 3DRSoC homogeneous whose the FPGA is a 2D Bloc Area model and the 3DRSoC heterogenous whose the FPGA is a 2D heterogeneous reconfigurable resources model.

The first section presents a spatio-temporal scheduling strategy, called 3D Spatio-Temporal Scheduling algorithm (3DSTS), for the 3DRSoC homogeneous system. 3DSTS consists in considering the 3rd dimension during the scheduling step in order to minimize the

global communication cost of the application. The second section presents a spatio-temporal scheduling strategy, called 3D HW/SW algorithm with SW execution Prediction (3DHSSP), for the 3DRSoC heterogenous system. The objective of 3DHSSP is to minimize the overall execution time of the application. 3DHSSP exploits the presence of processors in the MPSoC layer in order to anticipate a SW execution of a task when needed. Finally, the summary comes to conclude the chapter.

5.1 Considering communication cost in spatio-temporal scheduling for 3D Homogenous Reconfigurable SoC

5.1.1 Introduction

One of the main advantages of a system as 3DRSoC is the communication wire reduction. Apart the communications in horizontal direction, the vertical links allow tasks to exchange data between them easier and faster from a layer to another. This section addresses the issue of task scheduling and placement for a 3D Homogeneous RSoC. Each task executed on such system is supposed to be composed of two parts: the SW part and the HW part. The SW part is a portion of the code executed on a processor and the HW part is a function synthesized and configured in the FPGA. This composition enables the use of the FPGA as an accelerator for the SW execution. The spatio-temporal scheduling consists then in defining the scheduling time for each task and the placement for the two parts of each task on two layers of the 3D Homogeneous RSoC. Physical locations of SW parts and HW parts can have a significant impact on the delays during data transfers, which are directly linked to the communication cost of the system. Therefore, our main contribution in this section is to consider the communication cost during the task scheduling by evaluating not only the horizontal communication between tasks on each layer but also the vertical communication between two parts of a task from a layer to another layer.

5.1.2 Plateforme and task description

5.1.2.1 3D Homogeneous RSoC Model

The 3D Homogeneous RSoC (Fig 5.1) is composed of two layers: the MPSoC layer and the 2D Bloc Area FPGA layer. Each processor in the MPSoC layer has its own local memory and communicate with other processors through 2D Grid NoC. Also, each processor core is connected to a PRR whose the area is assumed to be identical to a processor area. A PRR has its own local memory and can accommodate at most one HW part of a task at a time. This characteristic makes a PRR having the same functionalities as a processor. Therefore, the 3D Homogeneous RSoC can be seen as a 3D CMP that has been studied in many works.

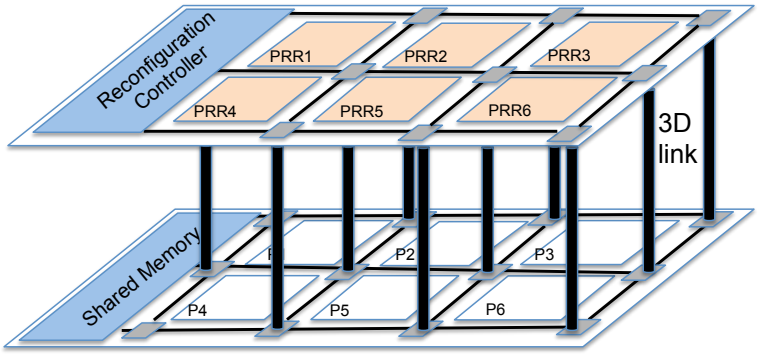


FIGURE 5.1: 3D Homogeneous RSoC

5.1.2.2 Task Model

As previously mentioned, we suppose that each task T_i in the application is composed of two parts: the SW part Ts_i and the HW part Th_i (see Fig 5.2(a) and Fig 5.2(b)). Th_i is used to accelerate the execution of Ts_i. These two parts always execute in parallel during the execution of T_i in order to support each other and exchange data between them. We call the communication ($Ts_i \Leftrightarrow Th_i$) the "vertical communication" of T_i (Fig 5.3). The amount of data exchanged between Ts_i and Th_i is noted $Csh_{i,i}$. The execution time of these two parts is the execution time of T_i, noted E_i.

When two tasks T_i and T_j must exchange data between them, the communication is supposed to be done at the end of the sender task and at the beginning of the receiver task. We call the communication between them the "horizontal communication" (Fig

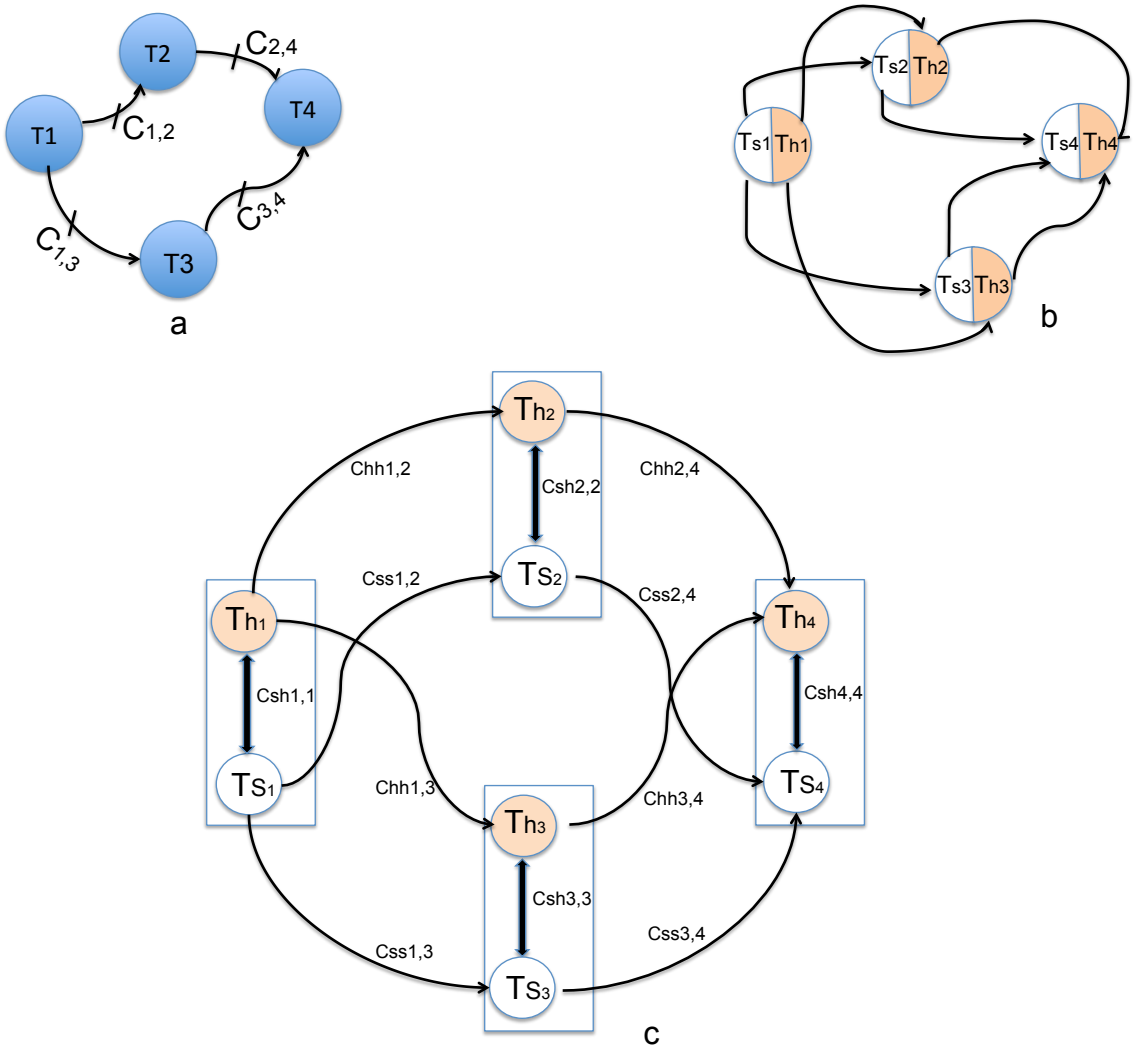


FIGURE 5.2: a - Classic model of the tasks graph ; b - The division of tasks in HW and SW parts ; c- The proposed task graph model

5.3). This communication is composed of: i) the communication whose the amount of data is noted $Css_{i,j}$, between the SW parts of these two tasks ($Ts_i \Rightarrow Ts_j$); ii) the communication whose the amount of data is noted $Chh_{i,j}$, between the HW parts of these two tasks ($Th_i \Rightarrow Th_j$). Figure 5.2(c) shows how a task set is modeled using our task model.

In a such system as 3D Homogeneous RSoC, the horizontal communication between two tasks can be done in a direct way or through the shared memory. In the case of direct communication, data are transferred from a processor to another processor (for the horizontal communication between two SW parts) and from one PRR to another PRR (for the horizontal communication between two HW parts) through the network on chip supported for each layer. In the case of memory communication, data are transferred to

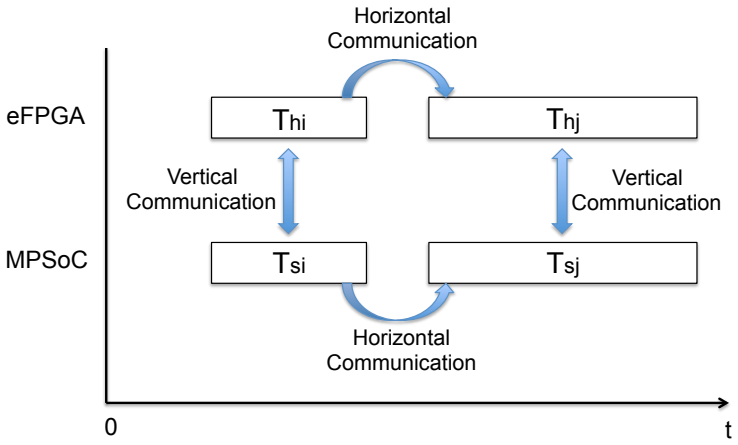


FIGURE 5.3: Task communication model for the 3D Homogeneous RSoC

the memory and at a later time, the receiver task retrieves data from the shared memory to process its execution. As in [3], Uniform Memory Access (UMA) technique is used for the communications through shared memory, i.e. the memory access time is independent of task positions.

In order to ease the lecture, we define three types of communications:

- Type 1: horizontal communication between the SW parts of two communicating tasks.
- Type 2: horizontal communication between the HW parts of two communicating tasks.
- Type 3: vertical communication between the SW part and the HW part of a task.

5.1.3 Communication Problem formalization

We analyzed in the chapter 3 that the communication through the shared memory leads to a high latency, thus can not guarantee the high-performance and high-bandwith requirement for running tasks on the FPGA. By favoring the direct communications, we can avoid the memory communications, thus the global communication cost of the application is minimized. Therefore, the problem of minimizing the global communication cost can be simplified as the problem of minimizing the total direct communication cost.

In this chapter, we keep all the definitions and assumptions made in the section 3.2.4 of the chapter 3 and extend the strategies applied in the chapter 3 to take into account

the 3rd dimension of the 3D homogeneous RSoC. Indeed, not only the communication between HW parts need to be considered but also the communication between SW parts and the communication between a couple (the SW part and the HW part) of a task. The formalization must then consider all possible communication types in order to minimize the total direct communication cost of the application.

Let G the task graph of the application, G is defined as

$$G = \{T, C\} \tag{5.1}$$

with T the set of tasks and C the set of communications (Fig 5.2(a)). The set of tasks is defined by

$$T \quad = \quad \{T_i\} = \{(Ts_i, Th_i)\} \qquad \forall\, i = 1, \ldots, N_T \tag{5.2}$$

with Ts_i and Th_i respectively the SW and HW parts of the task T_i (Fig 5.2(b)), E_i the worst case execution time of T_i and N_T the total number of tasks.

The set of communications is defined by

$$C = \{C_{i,j}\} \qquad \forall\, (i,j) = (1, \ldots, N_T), (1, \ldots, N_T) \tag{5.3}$$

with

$$C_{i,j} = \begin{cases} Css_{i,j} & \text{amount of data transferred } \textbf{from } Ts_i \textbf{ to } Ts_j \text{ (type 1) with i} \neq \text{j} \\ Chh_{i,j} & \text{amount of data transferred } \textbf{from } Th_i \textbf{ to } Th_j \text{ (type 2) with i} \neq \text{j} \\ Csh_{i,i} & \text{amount of data transferred } \textbf{between } Ts_i \textbf{ and } Th_i \text{ (type 3) with i} = \text{j} \\ Chs_{i,i} & \text{is supposed to be included in } Csh_{i,i}\text{, thus it is equal to 0 with i} = \text{j} \end{cases} \tag{5.4}$$

The distribution of tasks on the MPSoC layer and FPGA layer correspond to their physical locations on these two layers. As the communication cost between two parts depend on the distance and the amount of exchanged data between them, we propose first to model the distance as

$$\begin{cases} Dss_{m,n} = |Sx_m - Sx_n| + |Sy_m - Sy_n| \\ Dhh_{m,n} = |Hx_m - Hx_n| + |Hy_m - Hy_n| \\ Dsh_{m,n} = |Sx_m - Hx_n| + |Sy_m - Hy_n| \end{cases} \tag{5.5}$$

with $Dss_{m,n}$ the Manhattan distance between the processor S_m whose the physical barycenter position is (Sx_m, Sy_m) and the processor S_n whose the physical barycenter position is (Sx_n, Sy_n). $Dhh_{m,n}$ is the Manhattan distance between PRR_m whose the physical barycenter position is (Hx_m, Hy_m) and PRR_n whose the physical barycenter position is (Hx_n, Hy_n) . $Dsh_{m,n}$ is the Manhattan distance between the processor S_m and the PRR_n without considering the distance of the vertical link.

Then, the communication cost for different communication types can be modeled as

$$\begin{cases} CCss_{i,j,m,n} = \alpha * Dss_{m,n} * Css_{i,j} \ \ \forall(i,j) \\ CChh_{i,j,m,n} = \gamma * Dhh_{m,n} * Chh_{i,j} \ \ \forall(i,j) \\ CCsh_{i,i,m,n} = (\alpha * Dsh_{m,n} + \beta * L_{TSV}) * Csh_{i,i} \ \ \forall i \end{cases} \tag{5.6}$$

with $CCss_{i,j,m,n}$ the communication cost between Ts_i and Ts_j if Ts_i is executed on the processor S_m and Ts_j is executed on the processor S_n. α and γ are constants that model the communication times on the MPSoC layer and FPGA layer for a distance unit The constant β defines the transfer time for a distance unit on a TSV link and L_{TSV} is the vertical distance between the two layers. For this work, the value α, β and γ are set to 1. However, they can be adapted according to other 3D architectures of two or more than two layers).

The ultimate goal is to find an instantiation of the different parts of tasks (HW and SW) on the execution resources (Processors and PRRs), i.e. instantiation $As_{i,m}$ of task Ts_i on the processor S_m and instantiation $Ah_{j,n}$ of task Th_j on PRR_n which reduces total direct communication cost of the application. This goal can be written by the following compact form

$$Min \left(\sum_{i=1}^{N_T} \sum_{j=1}^{N_T} CCqr_{i,j,m,n} * Aq_{i,m} * Ar_{j,n} \right) \quad \forall\, q \in \{s,h\} \text{ and } \forall\, r \in \{s,h\} \tag{5.7}$$

with $Aq_{i,m} = 1$ if the task Tq_i is instantiated on the execution resource q (with $q = s$ for SW task and processor execution, or $q = h$ for HW task and PRR execution) or equal to 0 otherwise.

5.1.4 3D Spatio-Temporal Scheduling algorithm (3DSTS)

5.1.4.1 Strategy

The strategy evaluates the placement of receiver tasks at each sender task ending time so that the communication costs are limited as much as possible. In Chapter 3, we showed that the evaluation order of tasks to be scheduled is important and by evaluating the more costly communications first, i.e. a descending order of data exchange volume is chosen, the global communication cost will be reduced. The proposed strategy in Chapter 3 considered only the communication cost between HW tasks on a 2D Bloc Area FPGA. In this chapter, we extend the idea proposed in the chapter 3 by evaluating not only the horizontal communications between the sender task and the receiver tasks (type 1 and type 2) but also the vertical communication during the execution of receiver tasks. Thus, the more costly communications, among all communication types, will be chosen to evaluate first.

In this context, the goal of our algorithm consists in promoting the sequential placement at the same location of two parts which exchange a large number of data, when this placement is possible. So, for each communication type 1 (respectively type 2), the HW part (respectively the SW part) of the receiver task will replace the one of the sender task on the same PRR (respectively on the same processor). In theses cases, no transfer is needed between the two parts (data are stored in the internal memory of processor or PRR) and the communication cost can be considered equal to zero. When this replacement is not possible (execution resource not available), the objective is to place the two HW parts (or two SW parts) as close as possible to reduce the communication cost. The cost is directly related to the Manhattan distance between the two tasks on the same layer (for the type 1 and 2).

For the type 3, the data must be transferred between two layers through the vertical link (3D link). In this case, our algorithm tries to place the SW part and the HW part of the task in face-to-face to limit the distance and the communication delay. When this face-to-face placement is not possible, the placement must consider the location that reduces the distance between these two parts. Due to the strategy developed, our algorithm can produce solutions for which some couples of tasks (HW and SW parts) are vertically aligned and others are not aligned.

5.1.4.2 PseudoCode

The main part of our strategy is presented in the function 2. This function is executed at time t whenever one (or several) task T_i finishes its execution. We call this time point t the scheduling time.

At each scheduling time t, we build the communication list L containing all the horizontal communications (type 1 and type 2) from T_i to their receiver tasks T_j and the vertical communications (type 3) of T_j. Then, L is sorted in the descending order of data exchange amount to make sure the most costly communication will be evaluated first.

Before starting evaluating the communications in L, all the processors and the PRRs on which Ts_i and Th_i executed is set to be free (line 4-5), i.e. they are ready to accommodate other tasks. While the list L is not empty, we extract the first communication in L then try to place the receiver part of this communication so that the communication cost between the sender part and the receiver part is minimized. The function exits when L is empty.

This first communication in L can be type 1, 2 or 3. Each communication type is considered and the algorithm ensures the following placement:

- **Type 1 ($Ts_i \Rightarrow Ts_j$)**: the communication is done between the SW part Ts_i and the SW part Ts_j (lines [10-17]) on the MPSoC layer. If the processor S on which Ts_i was executed is free, Ts_j is placed on the same processor S. No transfer is needed and the communication cost between these two parts is zero. If the processor S is not free, we try to find the nearest free processor from S to placed Ts_j. When the placement of Ts_j is done, the processor supporting Ts_j becomes now occupied. Consequently, it is not anymore necessary to evaluate, in the list L, the communications from other sender SW parts to Ts_j.
- **Type 2 ($Th_i \Rightarrow Th_j$)**: the communication is done between two different HW parts (lines [19-26]) on the FPGA layer. The same strategy as type 1 is applied for this communication, excepted that Th_i and Th_j are executed on PRRs instead of processors.
- **Type 3 ($Ts_j \Leftrightarrow Th_j$)**: the communication is done between SW part Ts_j and HW part Th_j of the same task T_j (lines [28-50]) through the vertical link. If the

Algorithm 2 This part is executed whenever a task (or several tasks) finishes its execution

```
1: function 3DSTS (t, {T_i}){
2: Establish the list L set of communications: {{Ts_i ⇒ Ts_j}, {Th_i ⇒ Th_j}, {Ts_j ⇔ Th_j}}
3: Build the list L which is the rearrangement of L in descending order of exchanged data amount.
4: Make free the processors and the PRRs on which {Ts_i} and {Th_i} executed
5: ⇒ {Proc[Assign[Ts_i]]} = 0 and {PRR[Assign[Th_i]]} = 0.
6:
7: while {L} ≠ ∅ do
8:    (T_*i ⇒ T_*j) = L.ExtractFirstElement
9:    Switch (T_*i ⇒ T_*j) {
10:     Case (Ts_i ⇒ Ts_j):
11:       if (Proc[Assign[Ts_i]] = 0) then
12:          Assign[Ts_j] = Assign[Ts_i]
13:       else
14:          Assign[Ts_j] = FindNearestPosition[Ts_i]
15:       end if
16:       Proc[Assign[Ts_j]] = 1
17:       L.Delete(Ts_* ⇒ Ts_j)
18:
19:    Case (Th_i ⇒ Th_j):
20:       if (PRR[Assign[Th_i]] = 0) then
21:          Assign[Th_j] = Assign[Th_i]
22:       else
23:          Assign[Th_j] = FindNearestPosition[Th_i]
24:       end if
25:       PRR[Assign[Th_j]] = 1
26:       L.Delete (Th_* ⇒ Th_j)
27:
28:    Case (Ts_j ⇔ Th_j):
29:       if  (Proc[Assign[Ts_j]] = 1) and (PRR[Assign[Th_j]] = 1) then
30:          L.Delete(Ts_j ⇔ Th_j)
31:          break;
32:       else if (Proc[Assign[Ts_j]] = 1) then
33:          if  (PRR[Assign[Ts_j]] = 0) then
34:             Assign[Th_j] = Assign[Ts_j]
35:          else
36:             Assign[Th_j] = FindNearestPosition[Ts_i]
37:          end if
38:          PRR[Assign[Th_j]] = 1)
39:          L.Delete(Th_* ⇒ Th_j)
40:       else if (PRR[Assign[Th_j]] = 1) then
41:          if  (Proc[Assign[Th_j]] = 0)  then
42:             Assign[Ts_j] = Assign[Th_j]
43:          else
44:             Assign[Ts_j] = FindNearestPosition[Th_i]
45:          end if
46:          Proc[Assign[Ts_j]] = 1
47:          L.Delete(Ts_* ⇒ Ts_j)
48:       else
49:          L.Reverse(L[0], L[1])
50:       end if
51:    }
52: end while
53: }
```

placement of these two parts has already been done (lines [29-31]), it is not anymore necessary to evaluate the communication between them. Therefore, ($Ts_j \Leftrightarrow Th_j$) will be deleted from the list L. In the case only Ts_j has been placed (lines [32-39]), the ideal would be to placed Th_j face to Ts_j if the PRR in front of processor S where Ts_i is executed, is free. If this processor S is not free, we try to find the nearest free PRR from the processor S. Once Th_j is placed, all communications from other sender SW parts to Th_j must be deleted from L. The same strategy is applied in the case only Th_j has been placed but not Ts_j(lines [40-47]). Finally, in the case none of Ts_j and Th_j has been placed, this communication will be evaluated at a later time, until at least Ts_j or Th_j is placed.

Once all the communications in L are evaluated, i.e. L is now empty, the function exits and waits until the next scheduling time to be called.

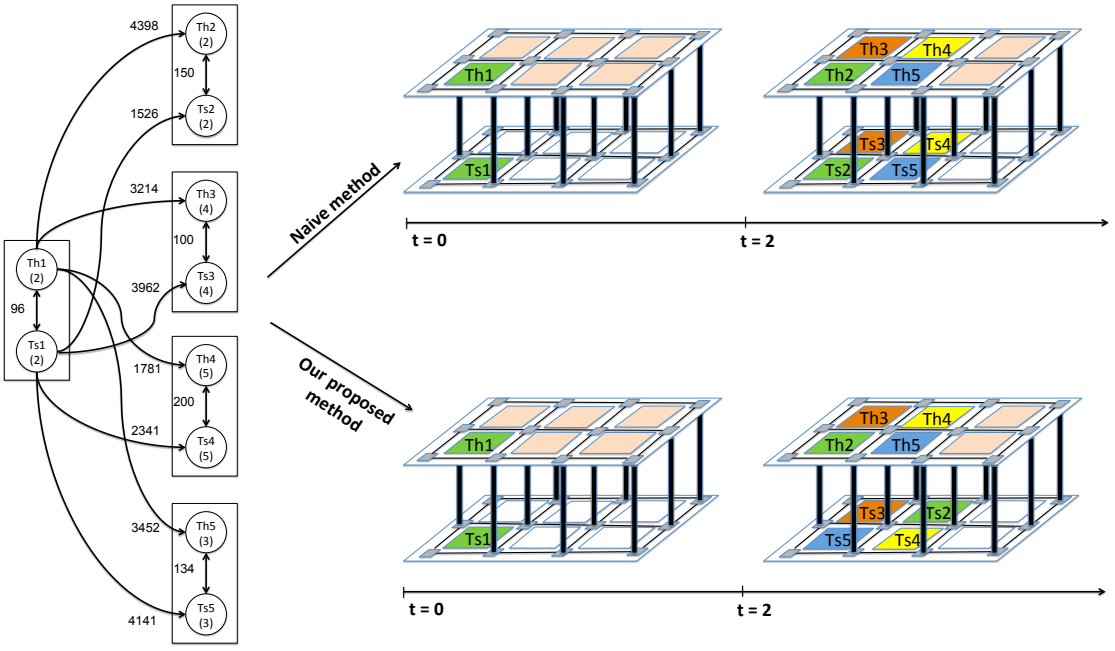


FIGURE 5.4: Placement solutions generated by the naive method and our proposed method

Fig 5.4 shows an example of the solutions generated by the native method and our proposed method. A part of the task graph is modeled as in the left of the figure. For the native method, when Ts_1 and Th_1 finish their execution (at time 2), the SW part of each receiver task will be placed in face-to-face of its HW part. For our proposed method, the placement decision can lead to place some couples of tasks face-to-face (for example

: Ts_3 and Th_3) and some others not aligned (for example: Ts_2 and Th_2, etc.) due to the different communication costs.

5.1.5 Evaluation

We compare our proposed strategy called 3DSTS with the one proposed in [3] where the placement of the SW and HW parts of a task is always face-to-face. We call R the ratio of data between the horizontal communications (type 1 or 2) and the vertical communications (type 3). In order to evaluate the comparison, we generate a large number of task graphs with different characteristics of R: i) R small ($R = 0.01$). ii) R close to 1. iii) R large ($R = 100$) as shown in Table 5.1.

	R = 0.01			**R = 1**			**R = 100**		
N_T	**(1)**	**(2)**	**G**	**(1)**	**(2)**	**G**	**(1)**	**(2)**	**G**
8	1516	1516	0	158709	177287	-11.70	232302	187429	19.31
9	3331	3331	0	116775	116775	0	354559	281300	20.66
10	3562	3562	0	203389	248252	-22.05	253741	213708	15.77
11	2809	2809	0	175685	175685	0	297718	258822	13.06
12	2280	2280	0	332643	379355	-14.04	505974	432284	14.56
13	4488	4488	0	233393	233393	0	700117	598539	14.50
14	5478	5478	0	500376	580998	-16.11	509579	430407	15.53
15	5612	5612	0	411398	498272	-21.11	709455	573457	19.16
16	4697	4697	0	259892	282872	-8.84	664138	542048	18.38
17	6630	6630	0	372977	376689	-0.99	848822	733665	13.56
18	5032	5032	0	436880	534535	-22.35	811384	694683	14.38
19	5482	5482	0	504933	585523	-15.96	930642	789348	15.18
20	6260	6260	0	584080	645004	-10.43	1059407	942412	11.04
21	5369	5369	0	393133	415781	-5.76	1254952	972175	22.53
22	8753	8753	0	486502	572204	-17.61	861478	726939	15.61
	Average		**0**	**Average**		**-11.13**	**Average**		**16.22**

TABLE 5.1: Assignment of tasks to a 3D Homogeneous RSoC platform simplified containing 4 processors and 4 PRRs. N_T: Total numbers of task. (1): Overall communication cost produced by the algorithm proposed in [3] . (2): Overall communication cost produced by our 3DSTS algorithm. G: gain of our algorithm compared to the one in [3] (%).

The results presented in Table 5.1 show the average gain of our algorithm compared with the one in [3] for different numbers of tasks. The 3D Homogeneous RSoC is composed of 4 processors and 4 PRRs. These results are graphically presented in Fig 5.5. Each task is composed of a HW and SW part, and the task graphs used are Direct Acyclic Graph (DAG), i.e. a direct graph with no direct cycles. We suppose that, except the root

tasks, each task has at least one predecessor. The characteristic values of tasks, number of dependencies or amount of exchanged data are randomly generated.

In the case i) $R = 0.01$ we randomly generated the amount of exchanged data from 100 to 500 data for each communication of type 1 (or 2) and from 10000 to 50000 for each communication of type 3. In the case ii) $R = 1$ the amount of exchanged data ranges from 10000 to 20000 for all of communication types. In the case iii) $R = 100$ we generated from 10000 to 50000 for each communication of type 1 (or 2) and from 100 to 500 for each communication of type 3. The execution time of each task ranges from 2 to 6 time units.

We note that when the amount of exchanged data for type 3 are more important than type 1 and type 2 ($R = 0.01$), our algorithm produces the same solution as the one in [3], i.e. the solution with the face-to-face placement between HW and SW part of a task. This result is not surprising due to the high cost of vertical communications which are privileged in the face-to-face placement.

When the amount of exchanged data for each type of communication are approximately equal (R close to 1), the communication cost obtained by the algorithm proposed in [3] algorithm could be less than our algorithm. This variation of communication cost is due to the small difference of amount of exchanged data between different type of communications. This difference may impact the face-to-face placement between HW part and SW part which increases the final communication cost.

When the amount of exchanged data for type 3 is less important than type 1 and type 2 ($R = 100$), our algorithm produces better solutions than the one in [3] (Fig 5.5). This is due to a high degree of possible choices in the strategy developed in our algorithm, while the possible choices in [3] are very limited.

We could always improve our algorithm by taking into account all values of R. This improvement can be done by comparing the communication cost produced by our algorithm and the one in [3] at each time t, whenever a task (or several tasks) finishes its execution. Then, the placement decision will follow the algorithm producing less cost.

However, the application with the small value of R or R close to 1 is not realistic in the context of our work. Indeed, a task with an important exchange between SW part and HW part means the flows of data go up and/or down to perform the processing. This

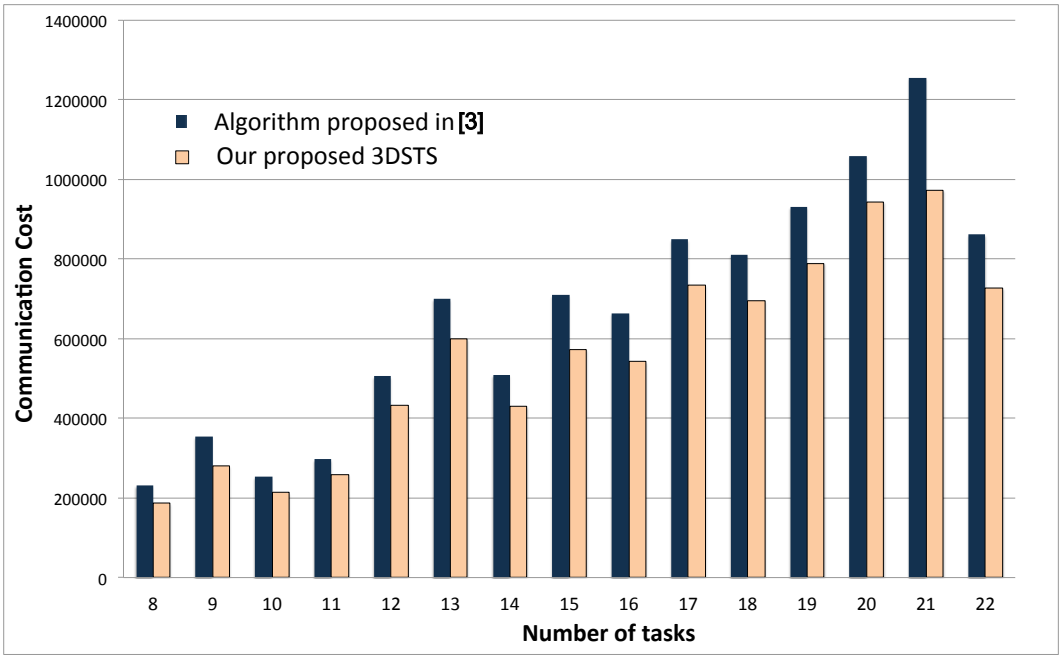


FIGURE 5.5: Comparison of the global communication cost generated by the algorithm proposed in [3] and our 3DSTS algorithm for the case $R = 100$

computation flow is not representative of the processing targeted for our architecture. If an important flow of data must be exchanged between several tasks, we think that this data flow will not change from the MPSoC layer to the FPGA layer (or vice-versa) after each task computation. Although our architecture can support this type of data flow, we suppose that the communications between layers will be more generally done to ensure control between SW part and HW part of a task. In this case, the signal controls are not much, and the ratio R will be generally high.

Another advantage of our algorithm compared with the one in [3] is the ability to fully exploit the execution resources when some tasks of application contain only a HW part or a SW part. In this case, our algorithm can place more tasks and has more choices for placing them into the different execution resources. Anyway, in this work, we have not addressed this aspect.

5.1.6 Conclusion

In this work, we have addressed the problem of spatio- temporal tasks scheduling for the 3D Homogeneous RSoC. The objective was to develop an algorithm that minimizes the overall communication cost of the application by taking into account all types of

communications between the tasks running on this platform. Our algorithm called 3DSTS evaluates, during the execution of the application, the need for the SW part and the HW part of a task to communicate in face-to-face through vertical links in order to minimize the global communication cost.

5.2 Considering execution time overhead in HW/SW scheduling for 3D Heterogeneous Reconfigurable SoC

5.2.1 Introduction

In the previous section, we considered the communication cost during the task scheduling for the 3D Homogeneous RSoC. The limitation from 3D Homogeneous RSoC is the constraints of the task model and the constraints of the 2D Bloc Area FPGA. For that, a task is composed of a SW part and a HW part which are always executed in couple (or in parallel). Then, due to the predefined PRRs, the resources occupied by the HW part of a task must be inferior than the area of a PRR.

In this section, the addressed architecture is the 3D Heterogenous RSoC whose the FPGA is a 2D heterogenous Reconfigurable Resources (RR) and more flexible to accommodate tasks of different sizes. This platform promises to provide a very high performance and flexibility through parallel and accelerated execution. It allows HW modules (also called HW tasks) to be executed on the FPGA while SW programs (also called SW tasks) to be executed on a processor of MPSoC. In this section, a different task model is considered when a task can be executed as a SW version as well as a HW version. These two versions are independent and only one of them can be executed at a time. For applications decomposed into a set of this kind of tasks, one of the main challenges is to determine on the fly which tasks should be run in HW and/or in SW, at which time, on which processor or in which region of the RR to achieve the best performance.

Therefore, our objective is to propose a HW/SW scheduling algorithm which manages tasks in time and in space on the 3D Heterogeneous RSoC platform to minimize the overall schedule time of the application. For that, we improve the previous work in Chapter 4 for HW task scheduling for the 2D heterogeneous FPGA and propose to exploit the presence of processor in the MPSoC layer in order to anticipate the SW

execution of a task if needed. The main idea is: for a task, our algorithm privileges the execution of its HW version, but executes its SW version in the case the SW version is ready and the HW version can not be allocated to RR (due to the unavailability of the internal configuration access port ICAP or due to the unavailable region for the task). Then, for tasks executed in SW, our algorithm will evaluate during the SW execution, the interest to continue the current SW version or to cancel it for starting the HW version. No preemption, no migration and no context saving are used in our method. When the scheduler decides to cancel the SW execution to start the HW version, all previous data produced until this time by the SW execution are deleted, then the HW version will start from the initial state.

5.2.2 Plateforme and task description

5.2.2.1 3D Heterogeneous RSoC Model

The 3D Heterogeneous RSoC is inspired from the 3D stacked chip of the ongoing Flextiles project which is composed of a many-core layer and a heterogeneous FPGA layer (see Fig 4.1 on page 67). The many-core layer contains several General Purpose Processor (GPP) and Digital Signal Processor (DSP) cores communicating through a 2D-Mesh Network-on-Chip (NoC). The FPGA has different types of resources, which are symmetrically located at fixed positions on the layer: computing resources (configurable logic blocks - CLBs), memory resources (Blockram - BRAMs) for storing data during computations and Accelerator Interfaces (AIs). Each AI is vertically connected with a router of the NoC and is used to homogenize the control of accelerators by the processors. The AI enables data transfer from SW to HW execution resources and vice versa, it offers also various management services as configuration, control and debug. More details about the 3D Flextiles can be found in [79].

The architecture of the 3D Heterogeneous RSoC is supposed to be almost the same as the 3D Flextiles chip but the it is more simple by considering that the many-core layer contains only homogenous GPPs. We assume that a shared memory connected to the NoC is used for supporting communications among tasks. The overview of the 3D Heterogeneous RSoC with the FPGA and MPSoC layer is shown in Fig 5.6(b).

The FPGA layer is modeled in the same way as in Chapter 4, i.e. it is defined by

$$\begin{aligned} RR &= \{(W_{FPGA}, H_{FPGA}), (Fb_{x0}, Fb_{y0}), (Fb_{\Delta x}, Fb_{\Delta y}), \\ &\quad (Fa_{x0}, Fa_{y0}), (Fa_{\Delta x}, Fa_{\Delta y})\} \end{aligned} \tag{5.8}$$

and the MPSoC layer is defined by N_p which is the total number of processors on MPSoC layer.

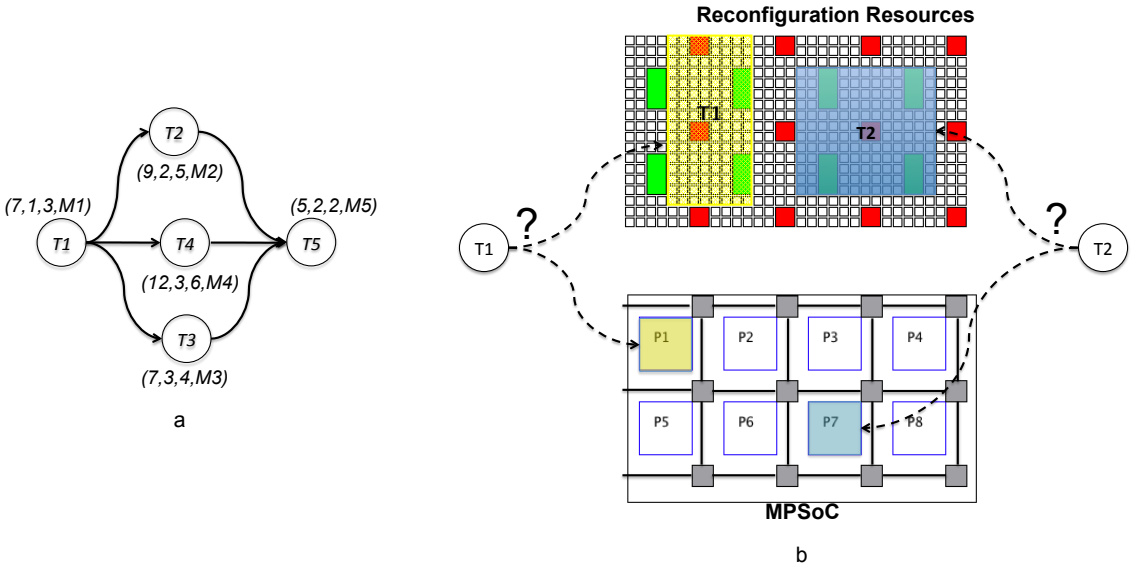


FIGURE 5.6: -a- Example of a task graph model; -b- Each task has the possibility to be run in SW or HW

5.2.2.2 Task Model

We represent a run-time application by a set of dependent tasks, the characteristics of each task are known at design time (before runtime) but not the execution flow. In our context, each task has both a SW implementation (or SW version) and a HW implementation (or HW version). The SW version of a task is a portion of code which is executable on a processor. The HW version of a task is a bitstream which is configurable on the FPGA to perform the task function. At design time, a task T_i is pre-characterized by four following parameters: i) SW worst case execution time (SE_i), ii) HW reconfiguration time (HR_i), iii) HW worst case execution time (HE_i), and iv) HW task model for T_i (M_i). M_i represents the resources required by T_i on RR. The communication time between tasks is considered being included in the worst case execution time of a task.

A task graph example of five tasks is shown in the figure Fig 5.6(a). T_1 is called the source task and T_5 sink task of the task graph. The edge between two tasks T_i and T_j signifies that T_j can only start its execution after T_i finishes its execution. As mentioned before, in dynamic scheduling, the scheduler does not have the information about the entire execution flow. The execution order is dynamically "discovered" during the task executions and is defined through a list containing ready tasks. In this work, we consider that each task has uniquely the knowledge about its predecessors and its successors. In this case, the scheduler can access to this information only when a task is requested. We note $Pred(T_i)$ (respectively $Succ(T_i)$) the list of predecessors (respectively successors) of tasks which produce (consume) data for the task T_i.

Fig 5.6(b) shows an example where the SW version of T_1 can be executed on the processor P_1. Also, the HW version of T_1 can be run on the RR and it will occupy M_1 resources as in the figure.

5.2.3 Spatio-temporal HW/SW scheduling formalization

In this part, we introduce the formalization for scheduling and placement of tasks on the 3DRSoC. The objective function and the main constraints of task scheduling and placement are presented. We first define the following variables whose some of them were defined in Chapter 4.

- N_T: number of tasks in the task set
- N_P: number of processors on MPSoC layer
- t_{hsr_i}: start time of HW reconfiguration of T_i in RR
- t_{hse_i}: start time of HW execution of T_i in RR
- t_{hfr_i}: finish time of HW reconfiguration of T_i, i.e. $t_{hfr_i} = t_{hsr_i} + HR_i$
- t_{hfe_i}: finish time of HW execution of T_i, i.e. $t_{hfe_i} = t_{hse_i} + HE_i$
- t_{sse_i}: start time of SW execution of T_i
- t_{sfe_i}: finish time of SW execution of T_i, i.e. $t_{sfe_i} = t_{sse_i} + SE_i$

- $t_{confirm_i}$: "confirm" time for T_i, at this time, our algorithm decides whether the SW execution of T_i should continue or be canceled to start a new HW execution. An example for the $t_{confirm_j}$ is shown in the Fig 5.8(a) where $t_{confirm_3} = 6$.

- $t_{swpredict_i}$: time that T_i is supposed to finish if the SW execution of T_i continues until the end.

- $t_{hwpredict_i}$: time that T_i is supposed to finish if the SW execution of T_i is canceled to start a new HW execution (with reconfiguration). As mentioned earlier, a task is ready to be scheduled when and only when all its predecessors have been already allocated. As all information about time scheduling of $Pred(T_i)$ is already known, we can easily evaluate when T_i will finish its HW execution.

- $Ah_{i,t} = 1$ if T_i is present on RR at time t, 0 otherwise.

- $As_{i,t} = 1$ if T_i is present on a processor at time t, 0 otherwise.

- $Nh_{i,t} = 0$ if T_i has never been allocated to RR until this time t, 1 otherwise.

- $Ns_{i,t} = 0$ if T_i has never been allocated to a processor until this time t, 1 otherwise.

- $S_{R_{k,i,t}} = 1$ if there exists at least one feasible region $R_{k,i}$ for T_i at time t, 0 otherwise. R_{ki} is defined below.

- $S_{P_{l,i,t}} = 1$ if there exists at least one available processor $P_{l,i}$ for T_i at time t, 0 otherwise.

- $Z_i = 1$ if T_i is fully executed on RR, i.e. present on RR from $[t_{hsr_i}, t_{hfe_i}]$, 0 if T_i is fully executed on a processor, i.e. present on MPSoC from $[t_{sse_i}, t_{sfe_i}]$.

- $FR_t = 1$ if the reconfiguration controller is occupied at time t, 0 otherwise. We suppose that only one reconfiguration controller is available, i.e. only one task can be reconfigured in HW at a time.

Objective

The main objective consists in minimizing the overall execution time of the application

$$min\,(t_{global}) \tag{5.9}$$

The overall execution time is given by

$$t_{global} = max\left\{t_{hfe_i} * Z_i + t_{sfe_i} * (1 - Z_i)\right\} \qquad \forall\, i = 1, \ldots, N_T \tag{5.10}$$

with N_T the total number of tasks in the graph. This minimization must respect followings constraints:

Scheduling Constraints

First of all, the minimization must ensure that each task is able to be executed on the RR and on an available processor:

$$\exists\, R_{k,i} \vee P_{l,i} \mid 1 \leq S_{R_{k,i,t}} + S_{P_{l,i,t}} \leq 2 \qquad \forall i = 1, \ldots N_T \tag{5.11}$$

with $R_{k,i}$ a feasible region on the RR for T_i and $P_{l,i}$ an available processor for the T_i. $R_{k,i}$ is a rectangle which is defined as

$$R_{k,i} = \{Rx_k, Ry_k, w_i, h_i\} \tag{5.12}$$

with Rx_k and Ry_k the coordinates of the bottom left point of the rectangle, and w_i and h_i the width and the heigh of T_i which are also the width and the height of the rectangle.

HW Placement Constraints

The placement of a task T_i on the RR must respect the placement constraints presented in the section 4.3 of Chapter 4, a region $R_{k,i}$ is called the feasible region for T_i when and only when all resources in this region are available and match those in the task.

Start HW Reconfiguration Constraints

A task T_i is ready to be scheduled at time t on the RR when and only when: i) all predecessors of T_i which have been executed in HW must have finished their reconfigurations and ii) for all predecessors of T_i which have been executed in SW, the "confirm" time has been taken.

$$t \geq max(t_{hfr_j} * Z_j + t_{confirm_j} * (1 - Z_j)) \qquad \forall j \mid T_j \in Pred(T_i) \tag{5.13}$$

A task T_i is said reconfigurable at time t when and only when it satisfies the following conditions:

- At least one feasible position for T_i is found at this time, i.e.

$$\exists\, R_{k,i} \mid S_{R_{k,i,t}} = 1 \tag{5.14}$$

- The configuration controller at this time is free; i.e

$$FR_t = 0 \tag{5.15}$$

- T_i has not been allocated to a processor until this time or T_i is currently running in SW but the task will finish earlier if its HW version starts reconfiguring from this time t.

$$\begin{aligned} & (Ns_{i,t} = 0) \\ \vee \quad & ((As_{i,t} = 1) \wedge (t_{hwpredict_i} \leq t_{swpredict_i})) \end{aligned} \tag{5.16}$$

Several tasks can be ready to be scheduled in HW but only one can be reconfigured at a time. $t_{hsr_i} = t$ when the scheduler decides to reconfigure T_i at this time.

Start HW Execution Constraints

Due to the data dependencies, a task T_i is ready to be executed at time t when and only when it satisfies the following conditions:

- The reconfiguration of T_i finished, i.e.

$$t \geq t_{hfr_i} \tag{5.17}$$

- All predecessors of T_i have finished their executions

$$t \geq max(t_{hfe_j} * Z_j + t_{sfe_j} * (1 - Z_j)) \qquad \forall j \mid T_j \in Pred(T_i) \tag{5.18}$$

When a task T_i is ready to be executed in HW, it will be executed immediately without the scheduler decision, therefore $t_{hse_i} = t$. Once the HW execution finishes, the task exits from the RR.

Start SW Execution Constraints

A task T_i is ready to be executed at time t on the MPSoC when and only when it satisfies the following condition:

- T_i has not been mapped in HW

$$Nh_{i,t} = 0 \tag{5.19}$$

- At least one available processor for T_i is found

$$\exists\ P_{l,i} \mid S_{P_{l,i,t}} = 1 \qquad \forall i = 1, \ldots N_T \tag{5.20}$$

- All predecessors of T_i have finished their executions

$$t \geq max(t_{hfe_j} * Z_j + t_{sfe_j} * (1 - Z_j)) \quad \forall j \mid T_j \in Pred(T_i) \tag{5.21}$$

When a task T_i is ready to be executed in SW, it will be executed immediately without the scheduler decision, therefore $t_{sse_i} = t$. Once the SW execution finishes, the task exits from the processor. Several tasks can be ready to be run in SW and they can be run in parallel on different processors at the same time.

5.2.4 HW/SW algorithm with SW execution Prediction (3DHSSP)

5.2.4.1 Strategy

In Chapter 4, the proposed STSH algorithm demonstrated that by integrating prefetching technique while considering two factors: the priority of new tasks to schedule and the placement, the overall execution time is significantly reduced. However, STSH only targets HW task scheduling problem for the 2D heterogeneous FPGA. In order to also

exploit the presence of processors in the MPSoC layer of 3DRSoC, we propose in this section a new algorithm called 3DHSSP which improves the STSH algorithm by anticipating the SW execution of a task if needed.

3DHSSP conserves three following techniques used in STSH: i) task priority, ii) configuration prefetching and iii) furthest task placement. In task priority technique, when several tasks are ready to schedule, the task having the most HW execution time length is requested to schedule first. Configuration prefetching is a well known technique to hide the reconfiguration of tasks run in HW, thus a HW version of a task can be reconfigured as soon as possible even if the HW execution of this task cannot start immediately due to the data dependencies with other tasks. Furthest task placement is a technique used to reduce the resource fragmentation. In that, if a task is run in HW, it will always be allocated to the furthest feasible region from the RR center.

The main idea of our 3DHSSP algorithm is: As executing a task in HW is usually faster than in SW and because the HW reconfiguration of a task can be anticipated earlier than the SW execution thanks to "prefetching" technique (see the constraints for starting a HW reconfiguration and a SW execution in 5.2.3), our algorithm always tries to allocate the HW version of the task on the RR as soon as possible. Nonetheless, in some cases, due to the unavailability of ICAP or unavailable region at the time the task is requested, the waiting time for accommodating the HW version can be long. If the HW version has not been accommodated yet at the time the SW version becomes ready and executable on an available processor, our algorithm will start the SW execution on the processor.

However, we cannot guarantee that executing a task in SW at that time will give a shorter completion time than waiting until RR can accommodate the HW version. In order to ensure the shortest completion time, our algorithm evaluates four following cases which are also illustrated by Fig 5.7:

- Case 1: the RR can accommodate the HW version of the task for starting its reconfiguration before the SW execution of the task becomes ready. In this case, it is not necessary to evaluate the possibility for the task to be run in SW. The task is chosen to be run completely in HW until the end of its HW execution and then exits from the RR.

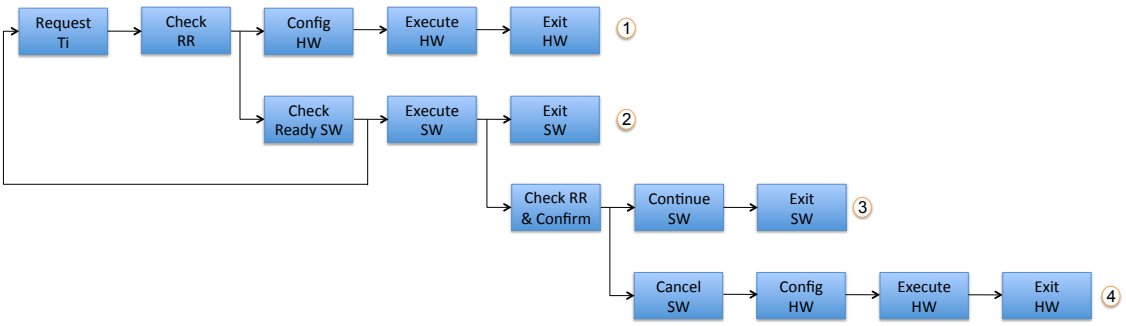


FIGURE 5.7: 4 possible happening scenarios during a SW execution

If the RR still cannot accommodate the HW version yet (due to the unavailability of ICAP and/or unavailable region of the RR) at the time the SW execution of the task becomes ready, our algorithm will start the SW execution of the task at that time. Then, during the SW execution, our algorithm will evaluate the cases 2, 3 and 4 presented in Fig 5.7:

- Case 2: at the end of the SW execution, the RR still cannot accommodate the HW version yet. In this case, the execution of the task is completed and the SW version exits from the processor. It is not anymore necessary to evaluate the possibility for the task to be run in HW.
- Case 3: during the SW execution, RR becomes available to accommodate the HW version but our algorithm realizes that continuing the SW execution will complete the task faster than canceling it for starting a HW version from the initial state. In this case, the SW execution will continue until it is completed and exit from the processor.
- Case 4: during the SW execution, RR becomes available to accommodate the HW version and our algorithm realizes that canceling the SW execution to start a HW version from the initial state will complete the task faster than continuing the SW execution. In this case, the SW execution will be canceled, the HW version will start configuring, then executing until it is completed and exit from RR.

For the case 3 and 4, the decision to continue or cancel the SW execution is done at the "Check RR & Confirm" state (Fig 5.7) , i.e. when the RR is able to accommodate the HW version of the task. At that time, our algorithm computes and compares two values: i) $t_{swpredict_i}$ which is the time the task T_i supposed to be completed if the running

SW of T_i continues until the end of its execution and ii) $t_{hwpredict_i}$ which is the time T_i supposed to be completed if the running SW of T_i is canceled to start the HW version. As the completion time of all predecessors of T_i is already known at that time, $t_{hwpredict_i}$ can be easily calculated based on this information. The final decision will follow the case giving a shorter value.

5.2.4.2 Example

Fig 5.8 shows the scheduling scenario for the task graph presented in Fig 5.6(a). Fig 5.8(a) shows how our algorithm performs from time slot 0 to time slot 6. In the figure, the green color represents a HW reconfiguration, the white color represents a HW execution and the purple color represents a SW execution. As mentioned earlier, due to the high SW execution cost, our algorithm always tries to allocate the task to RR whenever possible. Thus, T_1 is first allocated to the RR. By applying the prefetching technique, after the reconfiguration of T_1, its successors T_2, T_3 and T_4 are ready to schedule in HW. The priority of tasks is decided according to their HW execution time cost, thus T_4 which has the more costly HW execution time, is requested to be scheduled before T_2 and T_3. We suppose that feasible regions on RR for T_2, T_3 and T_4 exist at the slot time that they are requested to be scheduled. Therefore, T_4 is loaded at slot time 1.

At time 4, T_1 finishes its execution and data produced by T_1 for its successors are ready. At this time, T_4 finishes its reconfiguration and T_2 can start its reconfiguration. T_3 becomes ready to be executed in SW but it cannot be allocated to the RR due to the occupation of the configuration controller for T_2. Thus, our algorithm starts a SW execution of T_3 onto an available processor. This execution is a prediction that SW execution can reduce the overall execution time, but this decision must be canceled or confirmed later.

The important part of our algorithm comes from the time slot 6 when T_2 finishes its reconfiguration and the RR is available to accommodate the HW version of T_3. At this time slot 6 (that we define as $t_{confirm_3} = 6$), the algorithm must confirm or not the prediction of T_3 taken at time 4. For that, our algorithm will evaluate two cases: the case if T_3 continues until the end of its SW execution (in this case $t_{swpredict_3} = 11$ as in Fig 5.8(b)) and the case if T_3 cancels the SW execution to start its HW version (in this case $t_{hwpredict_3} = 13$ as in Fig 5.8(c)). As continuing the SW execution of T_3 is less time

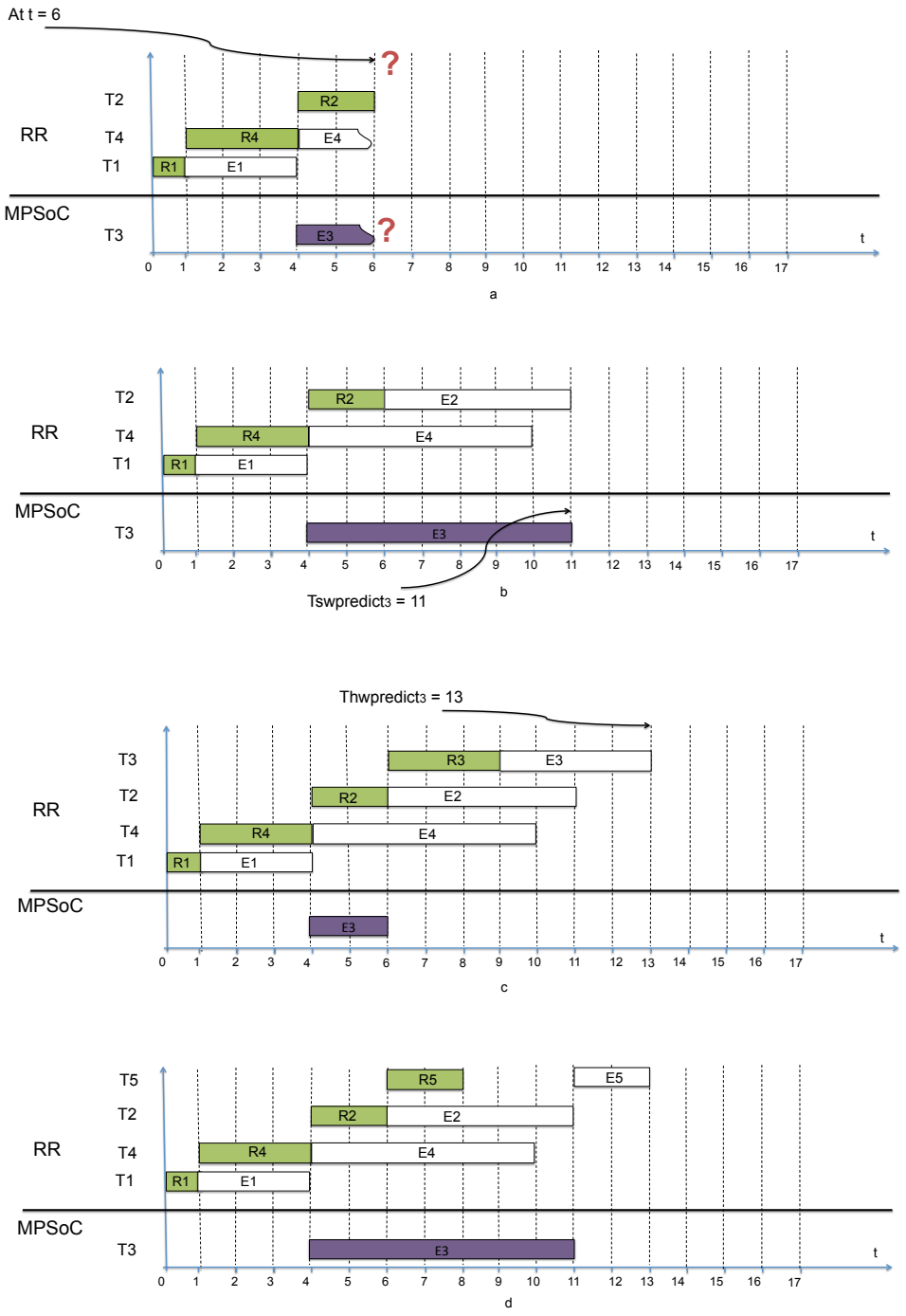


FIGURE 5.8: The scheduling scenario produced by our 3DHSSP stately

costly, our algorithm takes the final decision which is to continue the SW execution of T_3. The complete scheduling scenario proposed by our algorithm is showed in Fig 5.8(d).

5.2.4.3 Pseudocode

Algorithm 3 shows the pseudo-code of our algorithm. All constraints represented in the formalization part are respected in our algorithm. We call TL the list of tasks ready to be scheduled at time t. At the beginning (t = 0), TL contains only the source task. Other tasks will be discovered during the application execution and will be added in TL over time. Our algorithm will finish when and only when all the tasks in TL are scheduled and finished.

As mentioned previously, at time t, when T_i is requested, our algorithm tries to allocate T_i on the furthest feasible region from the center of RR whenever possible. `FindFurthestAvailableRegion` function (line 8) is used for searching this furthest region $R_{k,i}$. $R_{k,i}$ is represented as in equation 5.12 and it must respect the HW placement constraints. Lines [9-11] check whether the start HW reconfiguration constraints are respected to start reconfiguring T_i on RR (`StartHWAndUpdate`). If T_i cannot be allocated to RR at this time, the algorithm will check the start SW execution constraints (lines [22 - 25]) to start its SW execution on an available processor (`StartSWAndUpdate`). However, the SW execution of T_i can be continued or canceled later depending on the interest of its HW execution.

Then at a later time ($t_{confirm_i}$), when the configuration controller is free and the RR can receive the task, our algorithm will evaluate if it is better to continue the SW execution or cancel it to start the HW version (lines [12 - 18]). This evaluation is based on the comparison of the earliest time that T_i can finish, between $t_{swpredict_i}$ and $t_{hwpredict_i}$. If the condition 5.16 is satisfied, `CancelSWAndRelease` will cancel the SW execution of T_i on MPSoC and `StartHWAndUpdate` will start the HW reconfiguration on RR.

SS is a sublist containing the tasks whose the start time of HW reconfiguration and/or the "confirm" time is t. When a task is added to SS (line 20), its successors are also added in TL (line 27) and its successors will be ready to be scheduled at the next scheduling time. In order to reduce at maximum the overall execution time, `SortingHWE(`TL`)` is used to attribute the priority of tasks to schedule according to their HW execution time.

Algorithm 3 Pseudo-code of our proposed 3DHSSP algorithm

```
1: function 3DHSSP({Tasks}, 3DRSoC) {
2: t = 0
3: TL = {T_i}        | Pred(T_i) = ∅        ∀ T_i ∈ {Tasks}
4: while TL ≠ ∅ do
5:    SS = ∅
6:    Update(TL, RR, MPSoC)
7:    for all Ti ∈ TL do
8:       R_{k,i} = FindFurthestAvailableRegion(T_i)
9:       if (ReadyHW(T_i) & ( ∃R_{k,i} & (FR_t = 0 ))) then
10:         if Ns_{i,t} == 0 then
11:            StartHWAndUpdate (T_i, RR)
12:         else
13:            t_{confirm_i} = t
14:            t_{hwpredict_i} = CalculateEndHW(T_i)
15:            if t_{hwpredict_i} < t_{swpredict_i} then
16:               CancelSWAndRelease(T_i, MPSoC)
17:               StartHWAndUpdate (T_i, RR)
18:            end if
19:         end if
20:         SS = SS ∪ {T_i}
21:         TL = TL \ {T_i}
22:      else if ( ∃P_{l,i} & ReadySW(T_i)) then
23:         t_{swpredict_i} = t + SWE(i)
24:         StartSWAndUpdate (T_i, MPSoC)
25:      end if
26:   end for
27:   TL = TL ∪ newAvailableSuccesors (SS)
28:   SortingHWE (TL)
29:   t ← nextTimeSlot (t)
30: end while
31: return t
32: }
```

`Update(TL, RR, MPSoC)` (line 6) is used to remove a (or several) task(s) of TL from RR or MPSoC layer whether it has finished its execution at time t.

5.2.5 Evaluation

Comparison in terms of the overall execution time

To evaluate the robustness of our 3DHSSP algorithm, we generated different task sets with different number of tasks and compare the results produced by 3DHSSP with following algorithms:

- **Only SW (OSW)**: every task of the task set is allocated to the SW execution unit (processors).
- **Only HW (OHW)**: every task of the task set is allocated to the furthest feasible region of the RR. OHW extends the Napoleon algorithm [30] and takes into account the order of ready tasks to be scheduled according to their HE_i time.
- **First Come First Serve (FCFS)**: handles both HW and SW without "Check RR & Confirm" state, i.e. once a HW (or SW) task is run, it will be completely run until the end of its execution. In FCFS, the order of ready tasks is not sorted, the task which arrives first, is scheduled first.
- **Longest Execution Time First (LETF)**: handles HW and SW without "Check RR & Confirm" state, among ready tasks, the task having the longest HE_i is requested first.
- **Shortest Execution Time First (SETF)**: handles HW and SW without "Check RR & Confirm" state, among ready tasks, the task having the shortest HE_i is requested first.

The architecture defined for our tests is the 3DRSoC described in Fig 5.6(b), i.e. MPSoC contains 8 processors and RR is modeled as a rectangle whose the width is 32 and the height is 18 times a CLB size. Table 5.3 shows the results for 8 task sets which are generated according to the task characteristics in Table 5.2. N_T represents the total number of tasks in the task set. PD represents the parallelism degree, i.e. the number of successors that a task can have.

In every case, our 3DHSSP algorithm produces the shortest overall execution time (see table 5.3). To analyze the efficiency of our algorithm, for each example of N_T tasks, we analyze two factors: i) the number of confirmed decisions (noted N_{FS}), i.e. the number of tasks that started executing in SW and continued executing until the end of their SW execution, ii) the number of canceled decisions (noted N_{CS}), i.e. the number of tasks that started executing in SW but are then canceled to start its HW version.

As OSW algorithm allocates tasks only on processors, the overall execution time is high due to the SW execution time cost of each task. Compared to OHW algorithm, in the case that all the SW execution predictions are canceled (N_{CS} != 0 and $N_{FS} = 0$),

N_T	PD	HR_i	HE_i	M_i	SE_i
[8-15]	[1-4]	[10-30]	[50-100]	[7%-60%] of RR resources	[50-400]

TABLE 5.2: Task set characteristics with a low parallelism degree

	Examples of sets of tasks																					
	TS1			TS2			TS3			TS4			TS5			TS6			TS7			TS8		
OSW	1024			1237			1401			1535			1405			1508			1861			1516		
OHW	392			512			530			688			547			627			834			984		
FCFS	593			487			790			649			562			872			816			923		
LETF	513			426			637			781			562			741			1001			975		
SETF	593			654			790			649			574			666			1009			1128		
3DHSSP	**392**			**426**			**527**			**601**			**480**			**627**			**772**			**815**		
N_T	N_{CS}	N_{FS}	8	1	0	9	0	1	10	1	1	11	3	1	12	2	1	13	3	0	14	1	2	15	3	3

TABLE 5.3: Comparisons of the proposed algorithm with others in the case of task sets with a low parallelism degree [1-4]

the proposed 3DHSSP achieves the same results (see examples 1 and 6 of table 1). In other cases whenever at least one prediction is confirmed, our 3DHSSP is always better (see examples 2, 3, 4, 5, 7 and 8). Compared with other HW/SW scheduling methods as FCFS, LETF and SETF, our 3DHSSP always gives the shorter execution time. However, for the task sets with a low parallelism degree, the number of tasks executed in SW are not many. Thus, the overall execution time reduction is not much compared with OHW and other HW/SW methods.

N_T	PD	HR_i	HE_i	M_i	SE_i
[25-40]	[5-10]	[10-30]	[50-100]	[7%-60%] of RR resources	[50-400]

TABLE 5.4: Application characteristic with a high parallelism degree

To prove the efficiency of our algorithm in the case that the number of tasks and the parallelism in a task set increase, we generate 8 others task sets whose the characteristics are detailed in Table 5.4. The results shown in Fig 5.9 justify the advantages of our algorithm. As the parallelism degree is high, at a slot time, more tasks are requested to be allocated to processors and more "Check RR & Confirm" state must be taken. Therefore, OSW can better explore the parallelism capacity resulting a smaller difference between our 3DHSSP and OSW. However, compared with OHW and other HW/SW scheduling methods, our 3DHSSP shows clearly a better reduction than in the low parallelism degree case. In this case, our 3DHSSP allows achieving on average 42% reduction compared

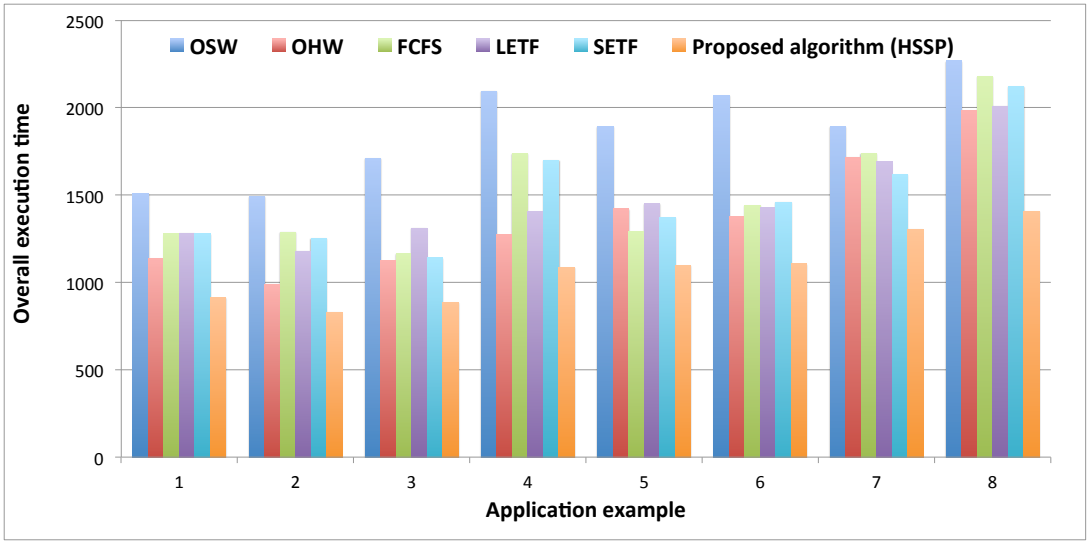


FIGURE 5.9: Comparisons of the proposed algorithm to others in the case of applications with a high parallelism degree [5, 10]

with OSW, 21% reduction compared with OHW and 26% compared with other HW/SW scheduling algorithms in terms of the overall execution time of the application.

Energy overhead

We assume that our method may increase the energy consumption of the system in several cases where the SW execution of a task T_i is canceled. In these cases, the SW execution of the task consumed energy for nothing. The time that a task starts executing in SW until it is canceled, is called "wasted time" ($Twasted_i = t_{confirm_i}$ - t_{sse_i}). In this work, even though our main objective is to minimize the overall execution time length of applications, we try to estimate the average energy overhead per processor to evaluate the efficiency of our algorithm. The "wasted energy" per processor is estimated by summing up the wasted time of all tasks in the application multiplied with the power active of processor (P_{active}), then dividing to the total number of processors:

$$\sum_{i=0}^{N_T} Twasted_i * \frac{P_{active}}{N_P} \tag{5.22}$$

The energy overhead is obtained by comparing the "waste energy" per processor with the sum of inactive time of processors multiplied with the standby power consumption of processor ($P_{standby}$). For the 8 previous examples having low parallelism degree and the 8 previous examples having high parallelism degree, the energy overhead is from 0% to 4% if $P_{active}/P_{standby} = 10$. This overhead seems acceptable in that case but it can

increase if the ratio $P_{active}/P_{standby}$ is higher. In the case where the ratio $P_{active}/P_{standby}$ is more important, a trade-off between the overall execution time length and the energy overhead must be found in order to give a better performance. It will be part of our future work.

5.2.6 Conclusion

In this work, we presented an online HW/SW task scheduling algorithm for the 3DRSoC. The objective of our proposed 3DHSSP is to minimize the overall execution time of a task set. 3DHSSP consists in evaluating, during runtime, the possibility to cancel the SW execution of a task to start a new HW execution of this task. It has been shown that our proposed algorithm reduces significantly the overall execution time of applications until 26% compared with other HW/SW scheduling methods. Minimizing the energy consumption and communication cost under the constraints of execution time will be analyzed in our future work.

5.2.7 Annex: Graphical Simulator

In order to graphically analyze the scheduling and placement solutions generated by our algorithms and some state-of-the-art algorithms, we have developed a generic tool that allows to couple a graphical interface with the developed algorithms. This tool helps us not only to analyze the results but also to debug and verify the proper functioning of the developed algorithms. The tool is developed in Qt/C++ by a student under my supervision.

Fig 5.10 shows how the tool looks like. The tool receives two following inputs: i) the architecture file describing the architecture model (the size of the FPGA, the positions of heterogeneous on the FPGA, the total number of processors, etc.). ii) the application file describing the task set model (the task characteristics, the dependencies between tasks, etc.). The zone 1 in the figure allows to choose the architecture and the application files. Depending on the application file, the tool generates, using the visualization software Graphviz, the task graph as in the zone 4. Also, depending on the architecture file, the architecture illustration is graphically generated (as in the zone 6). The zone 2 allows the user to choose the algorithm that we want to analyze (including our proposed

algorithms and some state-of-the-art algorithms). The buttons in the zone 3 allow the user to navigate to the desired time slot t and see what happen at this time. The zone 7 is reserved for displaying the information of current tasks on the 3DRSoC at this time. The zone 5 details the scheduling scenario from time 0 until t. The zone 6 details the task allocation on the RR layer, as well as on the MPSoC layer at this time and also at two previous time slots.

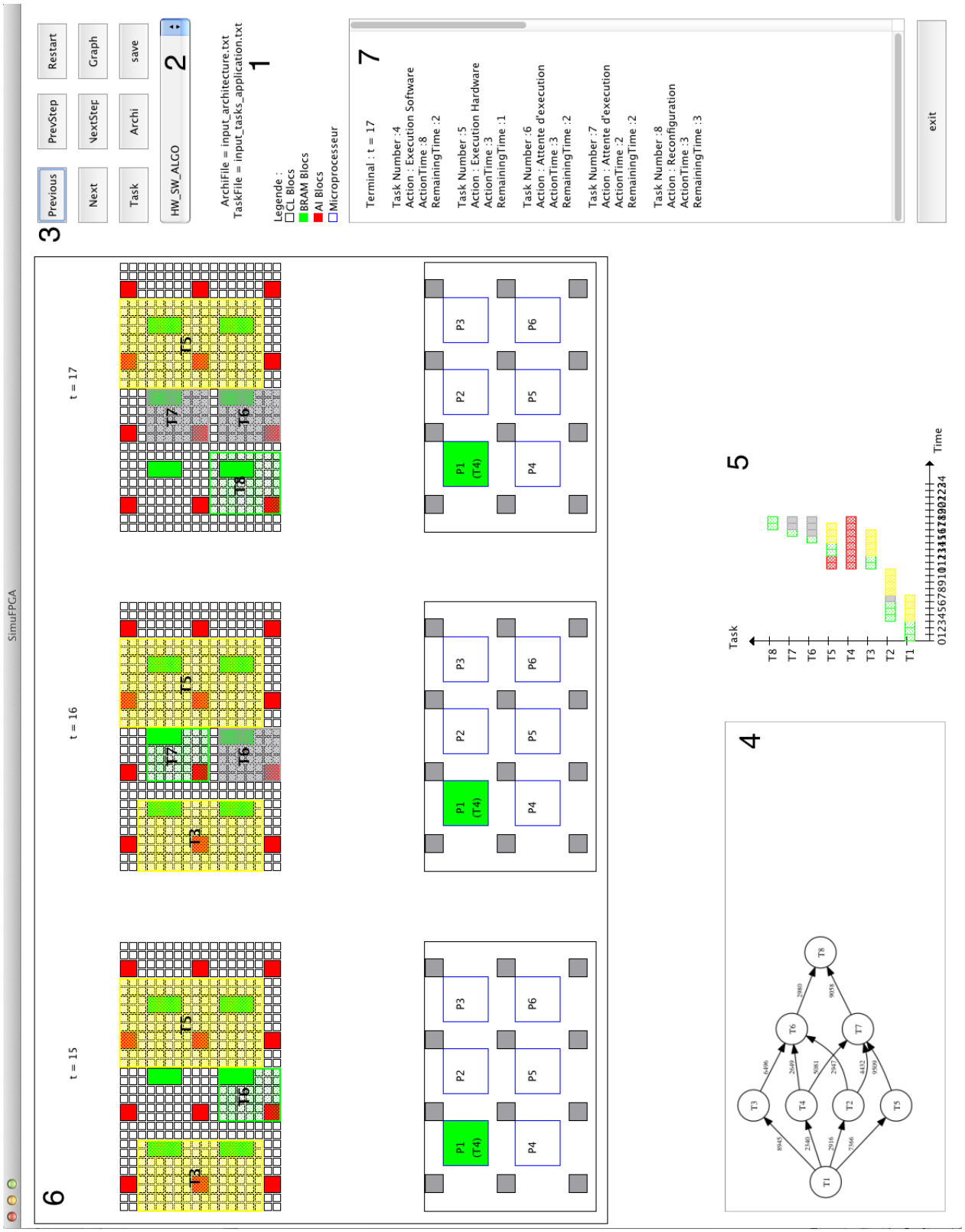


FIGURE 5.10: Simulator

5.3 Summary

In this chapter, we presented two different spatio-temporal scheduling strategies: one called 3DSTS for the 3D Homogeneous RSoC architecture and another called 3DHSSP for the 3D Heterogeneous RSoC architecture. 3DSTS considers, during the task scheduling, the third dimension of the architecture to place the couple of a task (the HW part and the SW part) in order to minimize the global communication cost of the system. Depending on the amount of exchanged data between tasks, 3DSTS decides the need to place the couple of a task in face-to-face through vertical links. The second algorithm 3DHSSP aims at minimizing the overall execution time of the application. 3DHSSP exploits the presence of processors in the MPSoC layer in order to anticipate a SW execution of a task when its HW version cannot be allocated on RR. Then, our algorithm evaluates, during the SW execution if it is better to stop the SW execution to restart the HW version of the task in the reconfigurable area. Even this technique is not efficient in energy consumption, we demonstrate that the execution time of an application can be reduced by applying the SW speculation.

Chapter 6

Conclusions and Perspectives

6.1 Conclusion

Stacking a MPSoC layer and a FPGA layer to form a 3DRSoC is a promising solution giving a high flexibility level in adapting the architecture to the targeted application. 3DRSoC allows to conserve the characteristics of a planar RSoC while inheriting all the advantages of the 3D technology. 3DRSoC allows software tasks to be executed on the MPSoC layer and hardware tasks to be dynamically allocated on the reconfigurable area. Thanks to the reduction of communication wires, the transfers between the layers are faster and better ensured. Other advantages expected from 3DRSoC compared with the planar RSoC is the manufacturing cost and the form factor reduction.

Even 3DRSoC can bring considerable benefits, several challenges need to be satisfied. One of them is the high-level management of tasks. For an application defined as a graph of parallel tasks running on the 3DRSoC, efficient scheduling and placement of tasks at different levels of the architecture on different components is extremely important to achieve high performance. In this thesis, our main goal is to propose online spatio-temporal scheduling strategies for two types of 3DRSoC: the 3D homogeneous RSoC and the 3D heterogeneous RSoC. These strategies must deal with two objectives: i) minimizing the global communication cost of the system and ii) minimizing the overall execution time of the application.

In such 3DRSoC architectures, the appearance of the FPGA layer supporting the dynamic and partial reconfiguration paradigm is crucial to achieve high performances and

high level of flexibility of the system. Compared to software tasks, the scheduling and placement of hardware tasks on the FPGA layer are more complex and should be taken more carefully into account. For example: a bad scheduling of hardware tasks (in time and space) can lead to an inefficient use of the reconfigurable area, thus future tasks can not be placed and the overall execution time of the application is penalized. Also, placing communicating hardware tasks far apart can have an important impact on the power consumption since they consume resources and routing signals. Therefore, before studying strategies for 3DRSoC architectures, it is essential to analyze the saptio-temporal scheduling strategies for the FPGA layer.

Regarding to the objectives of our work, the contributions of our works are following:

- Firstly, we analyzed the communication cost between hardware tasks executed on two homogeneous reconfigurable resources: the 2D Bloc Area FPGA and the 2D Free Area FPGA. We realize that by placing the communicating tasks as close as possible and by limiting the long communication with large amount of exchange data, the communication cost between tasks is reduced. Based on this principle, we propose Pfair Extension for Reconfigurable Resource **(Pfair-ERR)** strategy for the 2D Bloc Area FPGA and the Vertex List Structure Best Communication Fit **(VLS-BCF)** strategy for the 2D Free Area FPGA. Pfair-ERR is based on a well-known scheduling algorithm Pfair but we improve it to take into account the dependencies between tasks. Pfair-ERR produces interesting solutions with a low global communication cost but it produces also the preemption and migration costs which are non negligible in a reconfigurable architecture. VLS-BCF is based on Vertex List Structre (VLS) technique which is a low complexity and a simple data structure technique. VLS-BCF shows a significant reduction of communication cost compared to some existing methods as Kamer, FirstFit while not allowing the preemption and migration of tasks. This contribution is presented in Chapter 3.

- Secondly, we analyze the overall execution time of an application running on a 2D Heterogenous FPGA. We propose a spatio-temporal scheduling strategy called Spatio-Temporal Scheduling strategy for Heterogeneous FPGA **(STSH)** which allows to minimize the overall execution time of a task set executing on this 2D architecture. Our proposed heuristic integrates prefetching technique while considering two factors: the priority of new tasks to schedule and the placement decision

to avoid conflicts between tasks. The results show that our proposed heuristic significantly reduces the overall execution time compared to some non-prefetching and other prefetching methods. It also leads to a better resource utilization compared to others. This contribution is presented in Chapter 4.

- Finally, the main objectives and contributions are presented in the chapter 5 which extends the strategies proposed in chapter 3 and chapter 4 to take into account the 3rd dimension in a 3DRSoC. For that, not only the execution of hardware tasks on the FPGA but also the one of software tasks on the processors needs to be managed in an efficient way to exploit at maximum the advantages offered by 3DRSoC systems.

  In the first section of chapter 5, we propose the 3D Spatio-Temporal Scheduling **(3DSTS)** strategy for the 3D homogeneous RSoC. 3DSTS is based on the idea of Pfair-ERR and VLS-BCF. It consists in considering the 3rd dimension during the scheduling step to place the couple (the SW part and HW part) of a task in order to minimize the global communication cost. In 3DSTS, the placement decision can lead to place some couples of tasks face-to-face and some others not aligned due to the different communication costs. Compare with the native solution where the SW and HW parts of a task is always face-to-face, the 3DSTS offers a higher degree of flexibility for the spatial scheduling, thus leads to a significant reduction in global communication cost in almost application cases.

  In the second section of chapter 5, we present the 3D Hardware/Software with Software execution Prediction **(3DHSSP)** strategy for the 3D Heterogeneous RSoC. 3DHSSP improves the STSH strategy and exploits the presence of processor in the MPSoC layer in order to minimize the overall execution time of the application. In 3DHSSP, when a task must be scheduled and placed but no sufficient area on the FPGA is available to accommodate the task, its software version will be started. However, it is not sure that the software execution decision at this time can shorten the overall execution time of the application. Indeed, when a sufficient area becomes available to accommodate the task in hardware, 3DHSSP will evaluate whether the software execution must continue or whether it is better to stop the software execution to restart the task in the reconfigurable area. Even if this technique is not good for power consumption, we demonstrate that the execution time of an application can be reduced by applying this software speculation.

In this thesis, we address the spatio-temporal scheduling strategies for 3DRSoC architectures. Even if these platforms are still on progress and not yet available on the market, we have strong believes that our proposed strategies allow to analyze, evaluate and confirm the advantages expected from the 3DRSoC architectures. Then, developing these strategies in advance allows a shorter implementation time of the Operating System once these platforms are available.

6.2 Perspectives

In this section, we present three perspectives. For the continuation of my work, the implementation on a Zynq platform of an operating system supporting 3DSTSH strategy is on progress. Secondly, we propose to address the task scheduling and placement on a Virtex 7 platform which is composed of four FPGA layers horizontally connected. The advantages is the possibility to implement an operating system supporting the scheduling algorithms in a real system. Finally, we propose to integrate the fault management in the spatio-temporal algorithms. Due to the heterogeneity and the dynamic allocation of resources, some faults may appear during the execution of the application and they must be isolated in order to maximize the use of the FPGA resources.

Implementation of the 3DSTSH strategy

As mentioned in the introduction of this thesis, we did not address the implementation but only the simulation of the proposed algorithms. However, even if these 3DRSoC platforms don't exist, we can implement our strategies for the complex planar RSoC. The Zynq-7000 platform is an interesting planar RSoC in that we can implement an operating system supporting our proposed spatio-temporal scheduling algorithm, exceptionally the 3DSTSH algorithm. This Zynq platform has the ARM dual-core processors combining with a heterogeneous FPGA. Therefore, the 3DSTSH using the software speculation can be tested. The operating system can be installed on an ARM processor or on a softcore synthesized in the FPGA layer (microblaze) and it must be simple, open and supports the software/hardware task management.

Spatio-temporal scheduling strategies for Virtex 7 and implementation

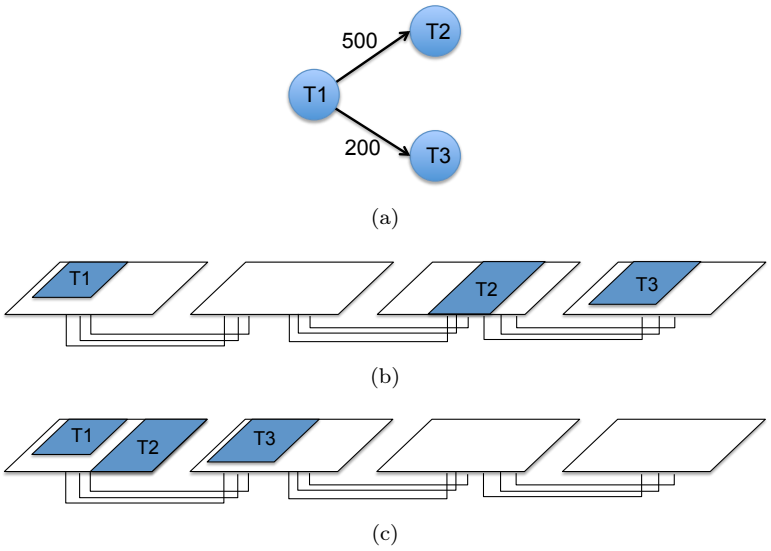


FIGURE 6.1: -a-example of three tasks with data dependencies; -b- Placing communicating tasks far apart; -c- Placing communicating tasks as close as possible

We plan to address the communication cost in spatio-temporal scheduling for a Virtex 7. As the Virtex 7 stacks four FPGA logic slices side-by-side and the allocation of tasks on different FPGAs can affect the communication cost between tasks, thus leading to the high latency and power consumption of the system. Considering three tasks with data dependencies as presented in Fig 6.1(a), placing T_2 and T_3 far from T_1 as in Fig 6.1(b) will lead to higher communication cost than placing T_1, T_2 and T_3 as in Fig 6.1(c).

In order to reduce the global communication cost of the system, our idea is to place communicating tasks in the same piece of FPGA and as close as possible. When the placement in the same FPGA is not possible, the receiver tasks will be placed in the nearest FPGA from the FPGA where sender tasks are located. For example in Fig 6.1(c), as T_2 receives more data from T_1, we favor to schedule T_2 before T_3. Then T_2 is placed close to T_1. The placement of T_3 cannot be done in the same FPGA due to the inadequate resources, thus T_3 is placed to the next FPGA.

The spatio-temporal scheduling needs to consider mainly the communication cost between tasks but also the FPGA fragmentation in order to able place communicating tasks in the same FPGA.

Two steps will be envisaged for this perspective: i) developing a spatio-temporal scheduling strategy which aims at reducing the global communication cost of the system and

comparing with other existing strategies. ii) implementing this strategy in a scheduling service of an operating system running on the Virtex 7.

Considering fault-isolation in spatio-temporal scheduling

During the execution of an application, it may happen that a (or several) fault occurs on the FPGA resources. Consequently, mapping a task on the region in which the fault is located is not acceptable. To tackle this problem, designers must provide solutions to manage faults when they appear in the system, in order to ensure successful execution of the application.

FIGURE 6.2: Different examples of task placement on the region containing a fault

Considering a simple example given in Fig 6.2. The FPGA is defined as a set of CLBs and we suppose that a fault occurs as in Fig 6.2(a). On this type of FPGA, when a fault

occurs, even if this fault affects one bit of register or one bit of a look-up-table (LUT), we consider that it affects a complete CLB.

At time t_1, when this fault is detected by the system during the execution of the task T_1, it means that the region R_1 covered by the task T_i contains a fault. However, we cannot detect exactly which resources are faulty. At the next time t_2, when T_2 must be placed, there is a probability that the system also detects a fault during the execution of T_2 if T_2 is placed on the region R_2 which overlaps with the region R_1. This probability of fault is high if the interconnection region between R_2 and R_1 is large and it is smaller if the interconnection region is small. If the region covered by T_2 is big as in Fig 6.2(c), a fault will surely occur during the execution of T_2. Otherwise, for another T_2 which is placed as in Fig 6.2(d), no fault is discovered during the execution of T_2, thus the region R_2 can be used for the next tasks. Fig 6.2(e) and 6.2(f) present the cases that a fault is detected during the execution of T_2.

To identify and isolate the faults, the spatio-temporal scheduling algorithm must integrate a model of probability for faults which is dynamically computed during the execution of the application. This probability is calculated based on the information of the regions where the faults are located and the interconnection regions between tasks. Depending on this probability, the spatio-temporal scheduling must decide where and when to place the coming tasks in order to quickly isolate the faults, thus maximize the available FPGA resources.

For this perspective, we propose two following steps: i) developing an accurate model of probability for faults and demonstrating in simulation that faults can be isolated during the execution of an application by using an efficient spatio-temporal scheduling strategy. ii) implementing the spatio-temporal scheduling strategy which supports this model of probability for faults in an operating system running on the FPGA.

Personal Publications

International Conference Proceedings

- **Quang-Hai Khuat**, Quang-Hoa Le, Daniel Chillet, and Sebastien Pillement. Spatio-temporal scheduling for 3d reconfigurable & multiprocessor architecture. In *8th International Design and Test Symposium (IDT)*, pages 1–6. IEEE, 2013.

- **Quang-Hai Khuat**, Daniel Chillet. Communication cost reduction for hardware tasks placed on homogeneous reconfigurable resource. In *International Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 265–270. IEEE, 2013.

- **Quang-Hai Khuat**, Daniel Chillet, and Michael Hubner. Considering reconfiguration overhead in scheduling of dependent tasks on 2d reconfigurable fpga. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 1–8. IEEE, 2014.

- **Quang-Hai Khuat**, Daniel Chillet, and Michael Hubner. Dynamic run-time hardware/software scheduling for 3d reconfigurable soc. In *International Conference on ReConFigurable Computing and FPGAs (ReconFig)*, 2014.

National Conference Proceedings

- **Quang-Hai Khuat**, Quang-Hoa Le, Daniel Chillet, Antoine Courtay, and Emmanuel Casseau. Ordonnancement spatio-temporel 3d minimisant le coût de communications entre tâches. In *GRETSI 2013*, pages 1–7, 2013.

Abbreviations

3DCMP	**3D C**hip **M**ulti**P**rocessors
3DHSSP	**3D H**ardware/**S**oftware with **S**oftware execution **P**rediction
3DIC	**3D I**ntegrated **C**ircuit
3DRSoC	**3D R**econfigurable **S**ystem-**o**n-**C**hip
3DSTS	**3D S**patio -**T**emporal **S**cheduling
AI	**A**ccelerator **I**nterface
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
BCF	**B**est **C**ommunication **F**it
BHF	**B**est **H**otspot **F**it
BL	**B**ottom **L**eft
BF	**B**est **F**it
BRAM	**B**lock **RAM**
CAD	**C**omputer-**A**ided **D**esign
CLB	**C**onfigurable **L**ogic **B**lock
CMP	**C**hip **M**ulti**P**rocessors
DAG	**D**irect **A**cyclic **G**raph
DPR	**D**ynamic **P**artial **R**econfiguration
DSP	**D**igital **S**ignal **P**rocessing
ECR	**E**xploited **C**omputation **R**esources
EDF	**E**arliest **D**eadline **F**irst
eFPGA	**e**mbedded **F**ield **P**rogrammable **G**ate **A**rray
F2B	**F**ace **T**o **B**ack
F2F	**F**ace **T**o **F**ace
FF	**F**irst **F**it
FPGA	**F**ield **P**rogrammable **G**ate **A**rray

GPP	**G**eneral **P**urpose **P**rocessor
HW	**H**ardware
ICAP	**I**nternal **C**onfiguration **A**ccess **P**ort
ILP	**I**nteger **L**inear **P**rogramming
I/O	**I**nput **O**utput
KAMER	**K**eep **A**ll **M**aximum **E**mpty **R**ectangle
LLF	**L**east **L**axity **F**irst
MER	**M**aximum **E**mpty **R**ectangle
MPSoC	**M**ulti-**P**rocessor **S**ystem-**o**n-**C**hip
MPRSoC	**M**ulti-**P**rocessor **R**Reconfigurable **S**ystem-**o**n-**C**hip
NER	**N**on-overlapping **E**mpty **R**ectangle
NPP	**N**earest **P**ossible **P**osition
NoC	**N**etwork-**o**n-**C**hip
NUMA	**N**on **U**niform **M**emory **A**cces
OS	**O**perating-**S**ystem
PBR	**P**artiall **B**istream **R**elocation
PoP	**P**ackage **o**n **P**ackage
PRM	**P**artially **R**econfigurable **M**odule
PRR	**P**artially **R**econfigurable **R**egion
RR	**R**econfigurable **R**esources
RSoC	**R**econfigurable **S**ystem-**o**n-**C**hip
SiP	**S**ilicon **i**n **P**ackage
STSH	**S**patio -**T**emporal **S**cheduling strategy for **H**eterogeneous FPGA
SW	**S**oftware
TSV	**T**hrough **S**ilicium **V**ias
UMA	**U**niform **M**emory **A**cces
VLS	**V**ertex **L**ist **S**tructure
VBS	**V**irtual **B**it**S**tream

Bibliography

[1] Yuan-Hsiu Chen and Pao-Ann Hsiung. Hardware task scheduling and placement in operating systems for dynamically reconfigurable soc. In *Embedded and Ubiquitous Computing–EUC 2005*, pages 489–498. Springer, 2005.

[2] Vincent Tong. The evolution of 3d ics leaping ahead of moore's law to deliver a 6.8b transistor device. *SiP Global Summit*, 2012.

[3] Quang-Hai Khuat, Quang-Hoa Le, Daniel Chillet, and Sebastien Pillement. Spatio-temporal scheduling for 3d reconfigurable & multiprocessor architecture. In *8th International Design and Test Symposium (IDT)*, pages 1–6. IEEE, 2013.

[4] Thomas Marconi, Yi Lu, Koen Bertels, and Georgi Gaydadjiev. A novel fast online placement algorithm on 2d partially reconfigurable devices. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 296–299. IEEE, 2009.

[5] Quang-Hai Khuat and Daniel Chillet. Communication cost reduction for hardware tasks placed on homogeneous reconfigurable resource. In *International Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 265–270. IEEE, 2013.

[6] Quang-Hai Khuat, Daniel Chillet, and Michael Hubner. Considering reconfiguration overhead in scheduling of dependent tasks on 2d reconfigurable fpga. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 1–8. IEEE, 2014.

[7] Quang Hai Khuat, Quang Hoa Le, Daniel Chillet, Antoine Courtay, and Emmanuel Casseau. Ordonnancement spatio-temporel 3d minimisant le coût de communications entre tâches. In *GRETSI 2013*, pages 1–7, 2013.

[8] Quang-Hai Khuat, Daniel Chillet, and Michael Hubner. Dynamic run-time hardware/software scheduling for 3d reconfigurable soc. In *International Conference on ReConFigurable Computing and FPGAs*, 2014.

[9] Joseph Y-T Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance evaluation*, 2(4):237–250, 1982.

[10] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[11] Aloysius K Mok. Fundamental design problems of distributed systems for the hard-real-time environment. 1983.

[12] Krishna Kavi, Robert Akl, and Ali Hurson. Real-time systems: An introduction and the state-of-the-art. *Wiley Encyclopedia of Computer Science and Engineering*, 2008.

[13] John M Calandrino, James H Anderson, and Dan P Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*, pages 247–258. IEEE, 2007.

[14] Sanjoy K Baruah, Neil K Cohen, C Greg Plaxton, and Donald A Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6): 600–625, 1996.

[15] Greg Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, and Scott Brandt. Dp-fair: A simple model for understanding optimal multiprocessor scheduling. In *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, pages 3–13. IEEE, 2010.

[16] Xuan Qi, Dakai Zhu, and Hakan Aydin. A study of utilization bound and run-time overhead for cluster scheduling in multiprocessor real-time systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*, pages 3–12. IEEE, 2010.

[17] G Umarani Srikanth, AP Shanthi, V Uma Maheswari, and Arul Siromoney. A survey on real time task scheduling. *European Journal of Scientific Research*, 69(1):33–41, 2012.

[18] Lodewijk T Smit, Johann L Hurink, and Gerard JM Smit. Run-time mapping of applications to a heterogeneous soc. 2005.

[19] Amit Kumar Singh, Thambipillai Srikanthan, Akash Kumar, and Wu Jigang. Communication-aware heuristics for run-time task mapping on noc-based mpsoc platforms. *Journal of Systems Architecture*, 56(7):242–255, 2010.

[20] Srinivasan Murali, Martijn Coenen, Andrei Radulescu, Kees Goossens, and Giovanni De Micheli. A methodology for mapping multiple use-cases onto networks on chips. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 118–123. European Design and Automation Association, 2006.

[21] Heikki Orsila, Tero Kangas, Erno Salminen, Timo D Hämäläinen, and Marko Hännikäinen. Automated memory-aware application distribution for multi-processor system-on-chips. *Journal of Systems Architecture*, 53(11):795–815, 2007.

[22] Fabrizio Ferrandi, Pier Luca Lanzi, Christian Pilato, Donatella Sciuto, and Antonino Tumeo. Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(6):911–924, 2010.

[23] Duncan Buell, Tarek El-Ghazawi, Kris Gaj, and Volodymyr Kindratenko. High-performance reconfigurable computing. *COMPUTER-IEEE COMPUTER SOCIETY-*, 40(3):23, 2007.

[24] Edson Horta and John W Lockwood. Parbit: a tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (fpgas). *Dept. Comput. Sci., Washington Univ., Saint Louis, MO, Tech. Rep. WUCS-01-13*, 2001.

[25] Heiko Kalte, Gareth Lee, Mario Porrmann, and U Ruckert. Replica: A bitstream manipulation filter for module relocation in partial reconfigurable systems. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 151b–151b. IEEE, 2005.

[26] Heiko Kalte and Mario Porrmann. Replica2pro: Task relocation by bitstream manipulation in virtex-ii/pro fpgas. In *Proceedings of the 3rd conference on Computing frontiers*, pages 403–412. ACM, 2006.

[27] Simone Corbetta, Massimo Morandi, Marco Novati, Marco D Santambrogio, Donatella Sciuto, and Paola Spoletini. Internal and external bitstream relocation for partial dynamic reconfiguration. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(11):1650–1654, 2009.

[28] Maamar Touiza, Gilberto Ochoa-Ruiz, El-Bay Bourennane, Abderrezak Guessoum, and Kamel Messaoudi. A novel methodology for accelerating bitstream relocation in partially reconfigurable systems. *Microprocessors and Microsystems*, 37(3):358–372, 2013.

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

[30] Francesco Redaelli, Marco D Santambrogio, and S Ogrenci Memik. An ilp formulation for the task graph scheduling problem tailored to bi-dimensional reconfigurable architectures. *International Journal of Reconfigurable Computing*, 2009:7, 2009.

[31] Javier Resano, Daniel Mozos, and Francky Catthoor. A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, pages 106–111. IEEE Computer Society, 2005.

[32] Yang Qu, Juha-Pekka Soininen, and Jari Nurmi. A parallel configuration model for reducing the run-time reconfiguration overhead. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 965–969. European Design and Automation Association, 2006.

[33] Yi Lu, Thomas Marconi, Koen Bertels, and Georgi Gaydadjiev. A communication aware online task scheduling algorithm for fpga-based partially reconfigurable systems. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 65–68. IEEE, 2010.

[34] Philipp Mahr and Christophe Bobda. Reducing communication costs on dynamic networks-on-chip through runtime relocation of tasks. In *Rapid System Prototyping (RSP), 2012 23rd IEEE International Symposium on*, pages 177–182. IEEE, 2012.

[35] Diana Göhringer, Michael Hübner, Etienne Nguepi Zeutebouo, and Jürgen Becker. Operating system for runtime reconfigurable multiprocessor systems. *International Journal of Reconfigurable Computing*, 2011:3, 2011.

[36] Christoph Steiger, Herbert Walder, and Marco Platzner. Heuristics for online scheduling real-time tasks to partially reconfigurable devices. In *Field Programmable Logic and Application*, pages 575–584. Springer, 2003.

[37] Christoph Steiger, Herbert Walder, and Marco Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *Computers, IEEE Transactions on*, 53(11):1393–1407, 2004.

[38] Michael Hubner, Christian Schuck, and Jürgen Becker. Elementary block based 2-dimensional dynamic and partial reconfiguration for virtex-ii fpgas. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–pp. IEEE, 2006.

[39] Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Design & Test of Computers*, 17(1): 68–83, 2000.

[40] Manish Handa and Ranga Vemuri. An efficient algorithm for finding empty space for online fpga placement. In *Proceedings of the 41st annual Design Automation Conference*, pages 960–965. ACM, 2004.

[41] Jin Cui, Qingxu Deng, Xiuqiang He, and Zonghua Gu. An efficient algorithm for online management of 2d area of partially reconfigurable fpgas. In *Proceedings of the conference on Design, automation and test in Europe*, pages 129–134. EDA Consortium, 2007.

[42] Yi Lu, Thomas Marconi, Georgi Gaydadjiev, and Koen Bertels. An efficient algorithm for free resources management on the fpga. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1095–1098. ACM, 2008.

[43] Ali Ahmadinia, Christophe Bobda, Marcus Bednara, and Jürgen Teich. A new approach for on-line placement on reconfigurable devices. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 134. IEEE, 2004.

[44] Lijun Wei, Defu Zhang, and Qingshan Chen. A least wasted first heuristic algorithm for the rectangular packing problem. *Computers & Operations Research*, 36(5):1608–1614, 2009.

[45] Jesús Tabero, Julio Septién, Hortensia Mecha, and Daniel Mozos. Allocation heuristics and defragmentation measures for reconfigurable systems management. *Integration, the VLSI Journal*, 41(2):281–296, 2008.

[46] Senoj Joseph Olakkenghil and K Baskaran. An fpga task placement algorithm using reflected binary gray space filling curve. *International Journal of Reconfigurable Computing*, 2014, 2014.

[47] Markus Koester, Mario Porrmann, and Heiko Kalte. Task placement for heterogeneous reconfigurable architectures. In *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, pages 43–50. IEEE, 2005.

[48] Antoine Eiche, Daniel Chillet, Sébastien Pillement, and Olivier Sentieys. Task placement for dynamic and partial reconfigurable architecture. In *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, pages 228–234. IEEE, 2010.

[49] Christophe Huriaux, Olivier Sentieys, Russell Tessier, et al. Fpga architecture support for heterogeneous, relocatable partial bitstreams. In *FPL-24th International Conference on Field Programmable Logic and Applications*, 2014.

[50] B Premalatha and S Umamaheswari. Survey of online hardware task scheduling and placement algorithms for partially reconfigurable computing systems.

[51] W Ye, Rolf Ernst, Th Benner, and Jörg Henkel. Fast timing analysis for hardware-software co-synthesis. In *Computer Design: VLSI in Computers and Processors, 1993. ICCD'93. Proceedings., 1993 IEEE International Conference on*, pages 452–457. IEEE, 1993.

[52] Rajesh K Gupta and Giovanni De Micheli. Hardware-software cosynthesis for digital systems. *Design & Test of Computers, IEEE*, 10(3):29–41, 1993.

[53] Peng Liu, Jigang Wu, and Yongji Wang. Hybrid algorithms for hardware/software partitioning and scheduling on reconfigurable devices. *Mathematical and Computer Modelling*, 58(1):409–420, 2013.

[54] Ahmed Al-Wattar, Shawki Areibi, and Faycal Saffih. Efficient on-line hardware/-software task scheduling for dynamic run-time reconfigurable systems. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 401–406. IEEE, 2012.

[55] Heiko Kalte and Mario Porrmann. Context saving and restoring for multitasking in reconfigurable systems. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 223–228. IEEE, 2005.

[56] Li Shang, Robert P Dick, and Niraj K Jha. Slopes: hardware–software cosynthesis of low-power real-time distributed embedded systems with dynamically reconfigurable fpgas. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(3):508–526, 2007.

[57] Jürgen Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, 2012.

[58] Stephen Tarzia. A survey of 3d circuit integration, 2008.

[59] Roshan Weerasekera, Li-Rong Zheng, Dinesh Pamunuwa, and Hannu Tenhunen. Extending systems-on-chip to the third dimension: performance, cost and technological tradeoffs. In *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*, pages 212–219. IEEE, 2007.

[60] Igor Loi, Federico Angiolini, and Luca Benini. Supporting vertical links for 3d networks-on-chip: toward an automated design and analysis flow. In *Proceedings of the 2nd international conference on Nano-Networks*, page 15. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007.

[61] Sudeep Pasricha. Exploring serial vertical interconnects for 3d ics. In *Proceedings of the 46th Annual Design Automation Conference*, pages 581–586. ACM, 2009.

[62] Mariam Sadaka, Ionut Radu, and Lea di Cioccio. 3d integration: Advantages, enabling technologies & applications. In *IC Design and Technology (ICICDT), 2010 IEEE International Conference on*, pages 106–109. IEEE, 2010.

[63] Peter Ramm, Armin Klumpp, Josef Weber, Nicolas Lietaer, Maaike Taklo, Walter De Raedt, Thomas Fritzsch, and Pascal Couderc. 3d integration technology: Status

and application development. In *ESSCIRC, 2010 Proceedings of the*, pages 9–16. IEEE, 2010.

[64] Kirk Saban. Xilinx stacked silicon interconnect technology delivers breakthrough fpga capacity, bandwidth, and xilinx stacked silicon interconnect technology delivers breakthrough fpga capacity, bandwidth, and xilinx stacked silicon interconnect technology delivers breakthrough fpga capacity, bandwidth, and power efficiency.

[65] Abbas Sheibanyrad, Frédéric Petrot, Axel Jantsch, et al. *3D integration for NoC-based SoC Architectures*. Springer, 2011.

[66] 3d ic industry summary. URL http://www.tezzaron.com/.

[67] Brent Goplen and Sachin Sapatnekar. Thermal via placement in 3d ics. In *Proceedings of the 2005 international symposium on Physical design*, pages 167–174. ACM, 2005.

[68] Xiuyi Zhou, Jun Yang, Yi Xu, Youtao Zhang, and Jianhua Zhao. Thermal-aware task scheduling for 3d multicore processors. *Parallel and Distributed Systems, IEEE Transactions on*, 21(1):60–71, 2010.

[69] Jiayin Li, Meikang Qiu, Jian-Wei Niu, Laurence T Yang, Yongxin Zhu, and Zhong Ming. Thermal-aware task scheduling in 3d chip multiprocessor with real-time constrained workloads. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2):24, 2013.

[70] Chiao-Ling Lung, Yi-Lun Ho, Ding-Ming Kwai, and Shih-Chieh Chang. Thermal-aware on-line task allocation for 3d multi-core processor throughput optimization. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[71] Marco Cox, Amit Kumar Singh, Akash Kumar, and Henk Corporaal. Thermal-aware mapping of streaming applications on 3d multi-processor systems. In *Embedded Systems for Real-time Multimedia (ESTIMedia), 2013 IEEE 11th Symposium on*, pages 11–20. IEEE, 2013.

[72] Jose Antonio Valero, Julio Septién, Daniel Mozos, and Hortensia Mecha. 3d fpga resource management and fragmentation metric for hardware multitasking. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–7. IEEE, 2009.

[73] Thomas Marconi and Tulika Mitra. A novel online hardware task scheduling and placement algorithm for 3d partially reconfigurable fpgas. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–6. IEEE, 2011.

[74] Thomas Marconi. Online scheduling and placement of hardware tasks with multiple variants on dynamically reconfigurable field-programmable gate arrays. *Computers & Electrical Engineering*, 40(4):1215–1237, 2014.

[75] Dalia Aoun, Anne-Marie Déplanche, Yvon Trinquet, et al. Pfair scheduling improvement to reduce interprocessor migrations. In *16th International Conference on Real-Time and Network Systems (RTNS 2008)*, 2008.

[76] James H Anderson and Anand Srinivasan. Early-release fair scheduling. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 35–43. IEEE, 2000.

[77] Christophe Bobda and Ali Ahmadinia. Dynamic interconnection of reconfigurable modules on reconfigurable devices. *Design & Test of Computers, IEEE*, 22(5): 443–451, 2005.

[78] Jukka Jylänki. A thousand ways to pack the bin-a practical approach to two-dimensional rectangle bin packing. *retrived from http://clb. demon. fi/files/RectangleBinPack. pdf*, 2010.

[79] Fabrice Lemonnier, Philippe Millet, Gabriel Marchesan Almeida, M Hubner, Jürgen Becker, Sébastien Pillement, Olivier Sentieys, Martijn Koedam, Shubhendu Sinha, Kees Goossens, et al. Towards future adaptive multiprocessor systems-on-chip: an innovative approach for flexible architectures. In *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pages 228–235. IEEE, 2012.

[80] C Claus, F Altenried, and W Stechele. Dynamic partial reconfiguration of xilinx fpgas lets systems adapt on the fly. *Xcell journal*, pages 18–23, 2010.

[81] Scott Hauck. Configuration prefetch for single context reconfigurable coprocessors. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 65–74. ACM, 1998.

[82] Zhiyuan Li and Scott Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 187–195. ACM, 2002.

Abstract:

Stacking a multiprocessor (MPSoC) layer and a FPGA layer to form a 3D Reconfigurable System-on-Chip (3DRSoC) is a promising solution giving a high flexibility level in adapting the architecture to the targeted application. For an application defined as a graph of parallel tasks running on this kind of system, one of the main challenges comes from the high-level management of tasks that is generally done by the scheduling service of the Operating System. Compare with the planar system, the scheduling problem on such 3D platforms is more complex due to the need to define, on the fly, what task should be run in software and/or hardware, when (temporal dimension) and where (spatial dimension, i.e. on what processor or what area of the FPGA) in order to exploit all the advantages of the 3rd dimension, thus achieve high performance of the system.

In this thesis, we propose online spatio-temporal scheduling strategies for 3DRSoCs. The first strategy decides, during the task scheduling, the need for a software and a hardware task to communicate in face-to-face so that the communication cost between tasks is minimized. The second strategy aims at minimizing the overall execution time of the application. It exploits the presence of processors in the MPSoC layer in order to anticipate, at run-time, the software execution of a task when its hardware version cannot be allocated to the FPGA. Eventually, a graphical simulation tool is developed to verify the proper functioning of the developed strategies and also enable us to produce results.

Résumé:

Empiler une couche multiprocesseur (MPSoC) et une couche de FPGA pour former un système sur puce reconfigurable en trois dimensions (3DRSoC) est une solution prometteuse donnant un niveau de flexibilité élevé en adaptant l'architecture aux applications visées. Pour une application exécutée sur ce type de système, l'un des principaux défis vient de la gestion à haut niveau de l'exécution des tâches. Cette gestion est généralement effectuée par le service d'ordonnancement du système d'exploitation et elle doit être en mesure de déterminer, lors de l'exécution de l'application, quelle tâche est exécutée logiciellement et/ou matériellement, à quel moment (dimension temporelle) et sur quelles ressources d'exécution (dimension spatiale, c-a-d sur quel processeur ou quelle région du FPGA) pour atteindre la haute performance du système.

Dans cette thèse, nous proposons des stratégies d'ordonnancement spatio-temporel pour les architectures 3DRSoCs. La première stratégie analyse l'intérêt de placer une tâche matérielle et une tâche logicielle en face-à-face afin que le coût de la communication entre ces tâches soit minimisé. La deuxième stratégie vise à minimiser le temps d'exécution global de l'application. Cette stratégie exploite la présence de processeurs de la couche MPSoC afin d'anticiper, au fil de l'eau, l'exécution d'une tâche logicielle quand sa version matérielle ne peut pas être allouée sur le FPGA. Finalement, un outil de simulation graphique a été développé pour vérifier le bon fonctionnement des stratégies développées et aussi nous permettre de produire des résultats.