Circuit partitioning for multi-FPGA platforms

Julien Rodriguez

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE
DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Julien Rodriguez**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Partitionnement de circuits pour plate-formes multi-FPGA

Soutenue le 6 septembre 2024

Devant la commission d'examen composée de :

M. Çevdet Aykanat ..	Professeur, Université de Bilkent	Rapporteur
Mme Lélia Blin	Professeure, Université Paris-Cité	Présidente
M. François Pellegrini	Professeur, Université de Bordeaux	Directeur
M. Viet Hung Nguyen .	Professeur, Université de Clermont-Auvergne	Examinateur
M. Dirk Stroobandt ..	Professeur, Université de Gand	Rapporteur

Invités - Encadrants :

M. François Galea	Ingénieur de Recherche, CEA	Encadrant
Mme Lilia Zaourar	Ingénieure de Recherche, CEA	Encadrante

Remerciements

Je tiens à remercier l'ensemble des personnes qui ont contribué, de près ou de loin, à l'élaboration de ma thèse de doctorat.

Je remercie Çevdet Aykanat et Dirk Stroobandt d'avoir accepté de rapporter mon manuscrit. Je remercie également Lélia Blin et Viet Hung Nguyen d'avoir accepté d'en être examinateurs.

Je souhaite profondément remercier mon directeur de thèse, François Pellegrini, qui m'a accompagné avec pédagogie pendant toute la durée de mon doctorat, tant sur des problématiques scientifiques que personnelles. Je remercie aussi l'équipe encadrante CEA, tout particulièrement François Galea qui, au-delà de ses conseils scientifiques, m'a enrichi dans d'autres domaines.

Mon doctorat avait la particularité d'être partagé entre deux sites : le CEA-List à Saclay et l'Inria à Bordeaux. Je remercie mes camarades de l'Inria pour les séminaires, les parties de baby-foot et de bowling, qui étaient d'agréables moments : rares, courts, mais intenses. Je remercie également, Caaliph, Fatma, Kods, et Mihail pour les riches discussions que nous avons eues.

Je tiens à remercier l'Association du CEA des thésards d'Île-de-France (ACTIF) pour les nombreux moments de distraction auxquels j'ai pu participer et pour la confiance que ses membres m'ont accordée en m'élisant vice-président. Pendant mon mandat, j'ai eu l'honneur d'organiser le voyage scientifique annuel, grâce à l'aide précieuse de mon camarade Adrien Roberty.

Je souhaite également remercier mes camarades Jonathan Fontaine et Valentin Gilbert, voisins de bureau, habitués des pauses cafés et du desk-basket. Jonathan, je me souviendrai toujours de ma première expérience en conférence internationale que nous avons partagée, riche en présentations et en casquettes. Je pense également à Simon Tollec et Pierre-Emmanuel Clet qui ont animé notre *open space*.

Comment ne pas mentionner et remercier également mes collègues de l'*open space* voisin : Samira Aït Bensaid, pour les discussions intéressantes sur les stratégies de recrutement ; Marc Renard, pour les apports en fruits, et Guillaume Roumage pour les entraînements athlétiques. Un remerciement spécial à Clément Fauchereau, amateur comme moi de poésies et de raisins.

Merci à Daniel Vert, Gabriella Bettonte, Lucas Phab et Valentin Gilbert pour les riches discussions en informatique quantique.

Je tiens à remercier la famille de ma compagne, qui m'a proposé à plusieurs reprises un espace très confortable pendant mes phases de rédaction en région Parisienne. Cathy, merci d'avoir été présente quand il fallait perdre au billard pour me remonter le moral, merci pour tout.

Je remercie fortement ma famille qui m'a accompagné avant, pendant et qui m'accompagnera sans doute après ce chapitre de ma vie, en particulier mes parents à qui je dédie l'ensemble de mon travail.

Résumé Un FPGA ('Field Programmable Gate Array') est un circuit intégré comprenant un grand nombre de ressources logiques programmables et interconnectables, qui permettent de mettre en œuvre, par programmation, un circuit électronique numérique tel qu'un microprocesseur, un accélérateur de calcul ou un système hybride complexe sur puce. Les FPGA sont largement utilisés dans le domaine de la conception de circuits intégrés, car ils permettent d'obtenir très rapidement des circuits prototypes, sans avoir à fabriquer la puce sur silicium. Cependant, certains circuits sont trop grands pour être mis en œuvre sur un seul FPGA. Pour résoudre ce problème, il est possible d'utiliser une plate-forme composée de plusieurs FPGA fortement interconnectés, qui peut être considérée comme un seul FPGA virtuel donnant accès à toutes les ressources de la plate-forme. Cette solution, bien qu'élégante, pose plusieurs problèmes. En particulier, les outils existants ne tiennent pas compte de toutes les contraintes du problème de placement à résoudre pour cartographier efficacement un circuit sur une plate-forme multi-FPGA. Par exemple, les fonctions de coût actuelles ne sont pas conçues pour minimiser les temps de propagation du signal entre les registres du FPGA, ni pour prendre en compte les contraintes de capacité induites par le routage des connexions. L'objectif de ce travail de doctorat est de concevoir des modèles de partitionnement et de placement d'hypergraphes adaptés au problème de placement des circuits sur une plate-forme multi-FPGA. Ces modèles seront spécifiquement conçus pour répondre aux objectifs et aux critères de performance définis par les concepteurs de circuits.

Mots-clés partitionnement, hypergraphe

Laboratoire d'accueil Centre d'intégration Nano-INNOV - CEA-LIST - Université Paris-Saclay, 2 Bd Thomas Gobert, 91120 Palaiseau

Title Circuit partitioning for multi-FPGA platforms

Abstract An FPGA ('Field Programmable Gate Array') is an integrated circuit comprising a large number of programmable and interconnectable logic resources, which allow one to implement, by programming, a digital electronic circuit such as a microprocessor, a compute accelerator or a complex hybrid system-on-chip. FPGAs are widely used in the field of integrated circuits design, because they allow one to obtain prototype circuits very quickly, without having to manufacture the chip on silicon. However, some circuits are too big to be implemented on a single FPGA. To address this issue, it is possible to use a platform consisting of several highly interconnected FPGAs, which can be seen as a single virtual FPGA giving access to all the resources of the platform. This solution, although elegant, poses several problems. In particular, the existing tools do not account for all the constraints of the placement problem to be solved in order to efficiently map a circuit onto a multi-FPGA platform. For example, current cost functions are not designed to minimize signal propagation times between FPGA registers, nor do they take into account the capacity constraints induced by the routing of connections. The aim of this PhD work is to design hypergraph partitioning and placement models adapted to the problem of circuit layout on a multi-FPGA platform. These models will be specifically designed to meet the objectives and performance criteria defined by circuit designers.

Keywords partitioning, hypergraph

Hosting Laboratory Centre d'intégration Nano-INNOV - CEA-LIST - Université Paris-Saclay, 2 Bd Thomas Gobert, 91120 Palaiseau

Contents

List of Figures

List of Tables

Résumé étendu en français

Un FPGA ("Field Programmable Gate Array") est un circuit intégré comprenant un grand nombre de ressources logiques programmables et interconnectables. Ces ressources permettent d'implémenter, par programmation, un circuit électronique numérique tel qu'un microprocesseur, un accélérateur de calculs ou un système hybride complexe sur puce. Les FPGA sont largement utilisés dans le domaine de la conception de circuits intégrés, car ils permettent d'obtenir des circuits prototypes très rapidement, sans devoir fabriquer la puce sur silicium. Cependant, certains circuits sont trop grands pour être implantés sur un seul FPGA. Pour résoudre ce problème, il est possible d'utiliser une plate-forme composée de plusieurs FPGA fortement interconnectés, qui peuvent être considérés comme un seul FPGA virtuel donnant accès à toutes les ressources de la plate-forme. Cette solution, bien qu'élégante, pose plusieurs problèmes. En particulier, les outils existants ne tiennent pas compte de toutes les contraintes du problème de placement à résoudre pour placer efficacement un circuit sur une plate-forme multi-FPGA. Par exemple, les fonctions de coût actuelles ne sont pas conçues pour minimiser le temps de propagation des signaux entre les registres, qui est pourtant crucial pour la performance du prototype résultant, ni ne prennent en compte les contraintes de capacité induites par le routage des connexions.

Le processus typique de conception de matériel électronique numérique comprend plusieurs étapes, incluant le prototypage, la vérification, le placement et le routage, qui peuvent concerner de très grands circuits logiques. Les méthodes mises en œuvre au cours de ces étapes tirent souvent parti d'approches de type « diviser pour régner », afin de séparer les circuits en sous-circuits de plus petites tailles. Ces sous-systèmes sont plus faciles à manipuler et visent à réduire le travail sur le circuit global.

Pour prototyper de grands circuits qui ne peuvent être implantés dans un seul FPGA, une plate-forme multi-FPGA est nécessaire. Dans ce cas, l'approche « diviser pour régner » est utilisée pour diviser les circuits en plusieurs morceaux, un pour chaque FPGA. Pour produire des partitions valables, il faut tenir compte des limites de capacité de chaque FPGA et des liens d'interconnexion. En outre, les partitions doivent éviter d'augmenter la longueur du chemin le plus long, appelé *chemin critique*. En effet, pour les circuits VLSI, la longueur du chemin critique détermine la fréquence maximale à laquelle le circuit peut fonctionner, et le placement de longs chemins sur plusieurs FPGA est susceptible de dégrader le chemin critique, du fait du temps de traversée plus long entre les deux composants.

L'objectif de cette thèse est de concevoir des modèles de partitionnement et de placement d'hypergraphes adaptés au problème de l'implantation de circuits sur une plate-forme composée de plusieurs FPGA. Ces modèles seront spécifiquement conçus pour répondre aux objectifs et aux critères de performance définis par les

concepteurs de circuits.

Le partitionnement d'hypergraphe multi-contraintes (MCHP) est couramment utilisé pour résoudre le problème de partitionnement de circuits et de prototypage. Dans ce contexte, les sommets de l'hypergraphe modélisent les broches des composants logiques et les hyperarêtes de l'hypergraphe représentent les fils qui les relient. Le problème de partitionnement équilibré des graphes est un problème NP-Difficile [125] pour lequel il n'existe pas de facteur d'approximation constant [11] à moins que $P = NP$.

Au cours des 30 dernières années, plusieurs outils de partitionnement d'hypergraphes ont été développés, tels que hMetis et son dérivé khMetis, PaToH et KaHyPar. Ces outils cherchent à minimiser la coupe (ou *min-cut*) entre les différentes parties calculées, cette coupe pouvant être mesurée selon différentes métriques.

Pour répartir les sommets entre les différentes parties, ces outils utilisent un *schéma multi-niveaux*, qui se compose de trois phases: *contraction*, *partitionnement initial*, et *raffinement*. La phase de contraction utilise récursivement une méthode de regroupement pour transformer le problème en un problème plus petit, qui possède les mêmes propriétés topologiques. Ensuite, un partitionnement initial est calculé sur le plus petit problème. Enfin, pour chaque niveau, la solution du niveau le plus grossier est prolongée jusqu'au niveau le plus fin, puis affinée à l'aide d'un algorithme d'optimisation locale. L'utilisation du schéma multi-niveaux permet de réduire le temps de calcul par rapport à une approche de partitionnement directe, les algorithmes de partitionnement les plus coûteux n'étant appliqués qu'aux hypergraphes les plus petits, alors que les algorithmes d'otimisation locale sont moins gourmands en ressources du fait qu'ils n'opèrent que sur une zone réduite des hypergraphes, à savoir la frontière entre les parties déjà trouvées.

État de l'art

De nombreuses approches ont été tentées pour améliorer les performances du partitionnement de circuits. Nous présentons ici quelques travaux récents sur le partitionnement de circuits pour le prototypage rapide qui prennent en compte les contraintes de performance. Beaucoup de ces travaux utilisent des outils de partitionnement *min-cut* existants, utilisés comme des boîtes noires, au sein d'algorithmes plus complexes, en pondérant les sommets et arêtes des hypergraphes afin de prendre en compte des contraintes supplémentaires que le partitionneur doit respecter.

Par exemple, dans [4], les auteurs présentent une approche multi-objectif basée sur hMetis. Les auteurs déterminent un ensemble fini de chemins les plus critiques à chaque étape de partitionnement, en utilisant un coût tenant compte de trois facteurs : la longueur du chemin critique, le nombre de fois où les chemins

de longueur critique sont coupés et le poids des hyperarcs associés aux chemins critiques.

Dans [38], les auteurs comparent une méthode classique utilisant hMetis pour le partitionnement, suivie d'un algorithme de placement, à une approche dérivée qui effectue du placement et du routage pendant l'étape de partitionnement. Les résultats produisent de meilleures valeurs de chemin critique par rapport à l'approche en deux étapes. Plus récemment, dans [127], les auteurs effectuent un pré-traitement et un post-traitement de l'hypergraphe considéré afin d'intégrer l'objectif de minimisation du chemin critique dans la métrique de la taille de coupe, en utilisant hMetis comme outil de partitionnement. Cependant, la minimisation de la taille des coupes n'est souvent pas l'objectif le plus pertinent. En outre, le fait de biaiser les fonctions de coût *min-cut* pour prendre en compte la minimisation du coût de chemins est insuffisant.

Contributions

Nos travaux portent sur le calcul du partitionnement équilibré d'hypergraphes avec minimisation de *coût du chemin critique*, en plus de l'objectif classique de la minimisation de coupe, qui est toujours pertinent pour nous car il contribue à réduire les contraintes de capacité de communication entre FPGA.

La première contribution de cette thèse est la définition d'une représentation spécifique des circuits électroniques numériques qui consiste en une union d'hypergraphes acycliques orientés (ou DAH, pour *directed acyclic hypergraph*) [160]. L'hypergraphe global représentant l'ensemble du circuit est supposé être connexe ; dans le cas contraire, ses composantes connexes peuvent être traitées indépendamment, aux capacités des FPGA près. Les sommets source et puits de chaque DAH sont étiquetés en rouge, tandis que les autres sommets sont en noir. Les sommets rouges représentent généralement des registres et des ports d'entrée sortie (E/S) et peuvent être partagés entre plusieurs DAH, ce qui rend connexe l'hypergraphe global. Les sommets noirs représentent des circuits combinatoires. Une fonction de coût des chemins modélise l'impact d'une coupe sur les chemins des DAHs pendant le partitionnement. Chaque partition d'un hypergraphe entraînera des coupures le long de certains chemins, induisant un coût de traversée supplémentaire. Notre objectif est de trouver une partition qui réduise la longueur maximale du chemin entre deux sommets rouges, qui correspond au temps minimum nécessaire pour calculer et sauvegarder les données de l'état du circuit dans les registres, et qui minimise également la taille de la coupe. Dans notre contexte, nous supposons que le coût de routage entre les parties puisse être non uniforme.

La plupart des méthodes actuelles de partitionnement de circuits sont basées sur des outils de partitionnement de graphes à usage général disponibles publique-

ment. Cependant, les outils de partitionnement actuels utilisent un modèle de coût qui n'est pas adapté au problème traité dans cette thèse. De plus amples détails sur les méthodes actuelles et leurs limites peuvent être trouvés dans le chapitre 2.

L'accent est d'abord mis sur les algorithmes de regroupement, c'est-à-dire les algorithmes dont l'objectif est de réduire la taille du problème. Ces algorithmes procèdent par fusion de sommets, c'est-à-dire que deux sommets deviennent un seul sommet ayant un poids égal à la somme des deux sommets regroupés. À ce titre, cette thèse propose une étude sur la pertinence du choix des sommets à fusionner. Dans cette étude, nous proposons de pondérer un couple de sommets en fonction du chemin critique local qui les traverse. Sur la base de cette stratégie de pondération, nous proposons une adaptation de l'algorithme de contraction d'hypergraphes *Heavy Edge Matching* (HEM), ainsi qu'un algorithme de regroupement de sommets (*clustering*), appelé *Binary Search Clustering* (BSC). Les résultats expérimentaux démontrent que l'algorithme BSC est plus performant que l'algorithme HEM sur notre modèle de pondération précité. La poursuite de nos travaux sur ce problème de regroupement concerne le rapport d'approximation paramétré par la taille maximum d'un groupe (*cluster*). Nous démontrons sous certaines hypothèses, une amélioration du rapport d'approximation, c'est-à-dire, le rapport entre le cout de la pire solution et la meilleure solution.

D'autres contributions de cette thèse concernent des algorithmes de partitionnement gloutons, présentés pour la première fois dans notre travail [160], et une approche de programmation en nombres entiers basée sur des contraintes d'ordonnancement tirées de notre travail [161]. Les algorithmes gloutons sont basés sur des algorithmes de parcours d'hypergraphe tels que le parcours en largeur et le parcours en profondeur. Nous avons aussi étudié un algorithme basé sur le calcule des composantes connexes. Afin de créer plusieurs composantes, nous fixons une taille à ne pas dépasser pour chaque composante. Dans nos travaux, nous tirons profit de notre schéma de pondération afin de guider l'algorithme de calcul des composantes connexes dans l'hypergraphes. Notons que ces algorithmes ne tiennent pas compte de la topologie cible. De fait, les résultats obtenus nous montrent que ces algorithmes sont performants vis-à-vis de l'état de l'art sur des topologies totalement connectées (graphe complet), mais moins efficaces pour d'autres topologies.

La suite de nos travaux s'est concentrée sur une extension de l'algorithme de recherche locale proposé par C. Fidducia et R. Mattheyses [75]. Notre algorithme étendu, appelé DKFM, permet d'optimiser la dégradation du chemin critique d'un circuit pour une partition passée en paramètre. Nous avons remarqué lors de nos expérimentations que notre algorithme DKFM permet de réduire la dégradation du chemin critique. Cependant, comme DKFM est un algorithme de recherche local, l'optimisation reste limitée vis-à-vis de partitions initiales calculées sans

tenir compte de la topologie cible.

Perspectives

Tous nos algorithmes ont été mis en œuvre en langage C, dans un cadre logiciel composé de structures de données pour représenter notre modèle d'hypergraphe rouge-noir, ainsi que des routines de service auxiliaires. Le problème de partitionnement adressé dans cette thèse diffère du problème de partitionnement traité par les outils hMetis, khMetis, KaHyPar et PaToH et, à notre connaissance, il n'existe pas d'outil public dédié à notre problème de partitionnement. Nous avons donc décidé de formaliser notre travail dans le logiciel RaiSin. Cet outil comprend un schéma de partitionnement composé des algorithmes présentés dans cette thèse, y compris une adaptation de notre algorithme de raffinement appelé DKFM. Notre algorithme de raffinement tient compte de la topologie mais peut ne pas être en mesure de fixer un partitionnement initial ne tenant pas compte de la topologie, car un algorithme de raffinement local n'est pas conçu pour reconsidérer les décisions globales. Par conséquent, la prise en compte de la topologie cible devrait être intégrée dans les algorithmes de partitionnement initial.

Notons également qu'il existe des biais d'approximation sur les chemins, crées pendant la phase de contraction. En effet, lorsque des sommets sont fusionnés ensemble, des faux chemins peuvent être crée et pris en compte. Par conséquent le maintient d'un cohérence sur chemins combinatoire doivent être étudiés et ajoutés au sein des algorithmes de contractions afin de faciliter le travail pour les algorithmes de partitionnement initiaux et de raffinement.

Introduction

In the context of prototyping digital circuits on a multiple Field Programmable Gate Array (FPGA), the circuit may be divided into smaller, manageable sections that can be implemented on separate Field-Programmable Gate Arrays (FPGAs) for testing and validation. This process is a crucial step in designing and testing large, complex digital systems.

Digital electronic circuits

A *digital electronic circuit* is a system of interconnected electronic components that manipulate discrete, binary signals to perform specific functions. Unlike analog circuits, which deal with continuous, varying signals, digital circuits operate using two discrete states: 0 (low) and 1 (high). These states represent binary logic levels and are used to represent information in a form that can be easily processed, stored, and transmitted by digital systems like computers. A digital circuit consists of several key components that collectively process binary signals, enabling boolean manipulation of information in electronic devices. *Logic gates* are the basic building blocks that perform basic boolean operations such as AND, OR and NOT, based on input signal values. *Flip-flops*, also denoted as *registers*, are used to store temporary data at fixed moments in time, determined by a global clock signal that periodically oscillates between 0 and 1. This is useful for both temporary storage and global synchronization between parts of the circuit. Therefore, digital electronic circuits are globally synchronized by the clock signal. Logic gates and flip-flops are the elementary blocks for constructing more complex structures: *counters* generate sequences of binary numbers, essential for applications such as timing and event counting. *Multiplexers* and demultiplexers make it easier to route signals to different destinations. *Arithmetic* and *Logic Units* (ALUs) perform arithmetic and logic operations. Memory units store data and instructions, distinguishing between volatile (such as RAM) and non-volatile (such as ROM) types. Microprocessors and controllers act as central processing units, executing instructions in digital systems. Together, these components enable electronic devices to process information with precision and reliability. A microprocessor or controller is an assembly of different types of units as described above.

Field-Programmable Gate Arrays

A Field Programmable Gate Array (FPGA) is a versatile electronic device that primarily consists of a matrix of programmable logic gates, a configurable interconnect network, programmable memories, clock management, input/output interfaces, a power management system. The logic gate array is the heart of the

FPGA, providing configurable blocks to perform various logic operations. Depending on the FPGA used, other elements may be present, such as hardwired multipliers or accelerators for neural network inference. Interconnects are used to link these elements and provide the necessary connections. Programmable memory resources are used to store data. The device also contains mechanisms for managing the clock signal, pins for I/O, a configuration block for storing parameters, and a power management system to ensure proper operation. Depending on the specific design, additional features such as specialized interfaces may be present for specific applications. These specific resources enable the implementation of complex digital electronic circuits by interconnecting them. FPGAs are reprogrammable after fabrication, allowing rapid design iterations and customization for specific tasks [28]. Their parallel processing capability, derived from an array of programmable logic devices connected by configurable routing resources, makes them exceptionally efficient for tasks requiring simultaneous operations. This feature is critical for accelerating computations in scientific applications, such as high-performance computing for simulation and scientific modeling. Their adaptability and programmability are essential in scientific research and development, enabling rapid prototyping and testing of novel algorithms and hardware designs. Overall, FPGAs are a cornerstone technology in modern digital electronics, with far-reaching implications for scientific and technological progress.

Circuit partitioning for rapid prototyping

Circuit partitioning consist in dividing the circuit into different parts. In the context of circuit prototyping, a circuit is synthesized and implemented on an FPGA. When a circuit is too large to fit into a single FPGA, a multi-FPGA platform may be used. The step of partitioning a digital circuit for a multi-FPGA platform for rapid prototyping is a critical methodology in digital system design. It involves breaking down a complex electronic circuit into smaller, manageable sections that can then be implemented on separate FPGAs for testing and validation. This process allows designers to distribute the computational load across multiple FPGAs, enabling the system to handle more complex tasks than a single FPGA could handle alone.

Partitioning must aim to optimize performance, resource utilization, and communication between FPGAs. Partitioning can potentially reduce circuit performance because communication between two FPGAs adds some amount of delay. Moreover, multi-FPGA platforms cannot be fully interconnected, so additional routing delay can also affect circuit performance. By carefully assigning specific modules or functions to different FPGAs, engineers can strike a balance between minimizing inter-FPGA communication and ensuring that each FPGA has suffi-

cient resources to perform its designated tasks effectively. This approach is particularly valuable when designing large-scale digital systems, such as high-performance computing applications, where the complexity and computational demands require the use of multiple FPGAs working together.

The effectiveness of the partitioning strategy will greatly influence the success of the multi-FPGA prototype.

A formal definition for the problem of circuit partitioning for multi-FPGAs platform can be found in Chapter 1, section 1.3.

Contributions

The first contribution of this thesis is the definition of the red-black hypergraph model, which is an extension of hypergraphs dedicated to the representation of digital electronic circuits. This representation allows us to better model physical constraints such as how partitioning affects performance, which is one of the main objectives addressed in this thesis. Based on this abstraction, we design a cost model and specific algorithms to solve the circuit partitioning problem for multi-FPGA platforms. Most current circuit partitioning methods are based on publicly available general-purpose graph partitioning tools. However, current partitioning tools use a cost model that is not suited for the problem addressed in this thesis. More details on current methods and their limitations can be found in Chapter 2.

Existing tools take advantage of the *multilevel framework*, which consists of three phases: *coarsening*, *initial partitioning* and *refinement*. The coarsening phase recursively uses a clustering method to transform the circuit model into a smaller one. During the second phase, an initial partitioning is computed on the resulting smaller circuit. Then, in the third phase, an algorithm is applied at each recursion level to prolong the computed circuit partition to the upper level and subsequently refine it. The multilevel framework reduces computation time compared to a direct partitioning approach. Computation time is important because partitioning a large circuit is a difficult and time-consuming problem. This is why this thesis presents algorithms for each stage of the multilevel framework.

The initial focus is on clustering algorithms, *i.e.*, algorithms whose goal is to reduce problem size. These algorithms proceed by merging elements, *i.e.*, two elements become a single element with a weight equal to the sum of the weights of the merged elements. For that matter, this thesis proposes a study on the relevance of the choice of the elements to be merged. Based on this study, an adaptation of the Heavy Edge Matching (HEM) algorithm is proposed, as well as a clustering algorithm with variable cluster size, called Binary Search Clustering (BSC). Experimental results demonstrate that BSC performs better than HEM on our circuit model. An improved result for the approximation complexity parameterized by

cluster size is also presented in this study on clustering algorithms [162].

Other contributions of this thesis concern greedy partitioning algorithms, Derived Breadth-First Search (DBFS) and Derived Depth-First Search (DDFS), both first introduced in our work [160]. DBFS and DDFS specialize the *breadth-first search* and *depth-first search* by adapting the traversal rules in order to tackle the purpose of this thesis. Alongside the study of greedy algorithms, we explore an integer programming approach based on scheduling constraints [161]. Later, we evidence that integer programming is difficult to implement on large instances. Finally, an extension of the Fidducia and Mattheyses refinement algorithm [75] for the problem addressed in this thesis is proposed in our work [160].

Outline

The following chapters present the main contributions of the thesis. Chapter 1 introduces the notations and definitions used in this dissertation. Our cost model and hypergraph structure are defined in this Chapter. Chapter 2 presents the current state-of-the-art in circuit partitioning. Chapter 3 introduces the experimental setup and the methodology to evaluate and compare our algorithms to the state-of-the-art. Chapter 4 introduces a weighting scheme and two clustering algorithms: an extension of heavy edge matching and the binary search clustering algorithm, and an approximation algorithm parameterized by cluster size. Chapter 5 presents greedy partitioning methods based on path algorithms, as well as an integer programming approach. Chapter 6 presents a solution refinement algorithm for the path length-aware partitioning problem.

Finally, a conclusion and perspectives for this work are presented in the last chapter of this thesis.

Appendix A.1 presents detailed experimental results obtained using the algorithms presented in the previous chapters and implemented in the open source RaiSin software. I have written RaiSin during my PhD studies and it is the testing ground for all of the algorithms that are presented in this thesis.

Chapter 1

Definitions

This chapter presents notations and definitions concerning graphs and hypergraphs, and operations attached to them. In Section 1.1 graphs, hypergraphs and the red-black hypergraph are defined.

The notion of criticality is presented in Section 1.2, and the problem of partitioning for path-length minimization to a target topology is defined in Section 1.3. A conclusion of this chapter can be found in Section 1.4.

1.1 Graphs and Hypergraphs

In this section, we define graphs, hypergraphs and notions attached to them. These mathematical objects will be used throughout this thesis, as support for proofs. The first paper on the use of graphs in a mathematical context is an article by the Swiss mathematician Leonhard Euler, published in 1741 [70]. This article deals with the famous problem of the "seven bridges of Königsberg". The problem was to find a way for a traveler to return to their starting point, by crossing each bridge exactly once. Hypergraphs are a generalization of graphs. They were first introduced by Claude Berge in his book *Graphes et Hypergraphes*, published in 1970 [21].

1.1.1 Graphs

Definition 1.1.1. *A graph* $G \stackrel{def}{=} (V, E)$ *is defined as a set of vertices* V*, and a set of edges* E*. An edge is an unordered set of vertex duplets.*

In this thesis, we consider the sets V and E to be finite. Given some graph G, we denote the set of vertices by $V(G)$ and the set of edges by $E(G)$. The number of vertices in a graph is called the *order* of the graph, noted $|V(G)|$. As $V(G)$ and $E(G)$ can be weighted, we will denote by W_V the set of vertex weights, and W_E the set of the edge weights. The weights can be multivalued.

Definition 1.1.2. *A directed graph* $G \stackrel{def}{=} (V, A)$ *is defined as a set of vertices* V*, and a set of arcs* A*. An arc is an ordered set composed of two vertices. The first vertex is the source and the second one is the sink.*

In this thesis, we consider the sets V and A to be finite. We denote by $A(G)$ the set of arcs of a directed graph G. As $V(G)$ and $A(G)$ can be weighted, we will define by W_V the set of vertex weights, and W_A the set of the arc weights. In the general case, we consider multivalued arc and vertex weights. Examples of a graph and a directed graph can be found in Figure 1.1.

An edge or an arc of the form $\{u, u\}$ is called a *loop*. Whenever an edge exists in several instances in the set of edges E, it is called a *multiple edge*. For an arc that exists in several instances in the set of arcs A, it is called a *multiple arc*.

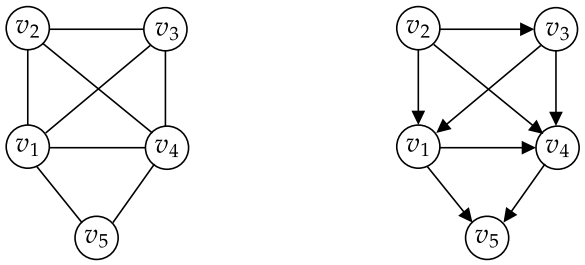


Figure 1.1: Examples of a graph and a directed graph.

For a vertex u, the set of its neighboring vertices $\Gamma(u)$ is defined as follows:

$$\Gamma(u) \stackrel{\text{def}}{=} \{v | u \neq v, \{u, v\} \in E\} \ . \tag{1.1}$$

An edge or an arc $\{u, v\}$, is said to be *incident* to vertex u and vertex v. Vertices u and v are the *ends* of edge or an arc, and are called *neighbors*.

According to the definition of an arc $a = \{u, v\}$, vertex u is considered to be an *incoming neighbor* of v, and v is considered to be an *outgoing neighbor* of u. Let u be a vertex; the set of its incoming neighbor vertices $\Gamma^-(u)$ is defined as:

$$\Gamma^-(u) \stackrel{\text{def}}{=} \{v | v \neq u, \{v, u\} \in A\} \ , \tag{1.2}$$

and; the set of its outgoing neighbor vertices $\Gamma^+(u)$ is defined as:

$$\Gamma^+(u) \stackrel{\text{def}}{=} \{v | v \neq u, \{u, v\} \in A\} \ . \tag{1.3}$$

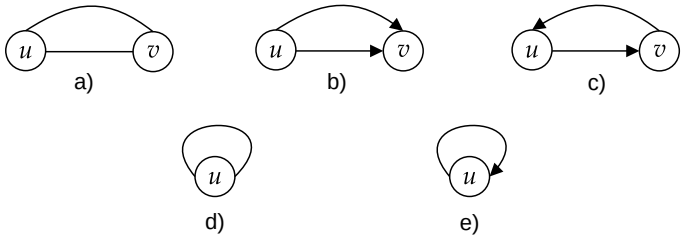


Figure 1.2: Examples of loops and multiple arcs and edges.

Figure 1.2 shows several examples of loops and multiple edges or arcs. Example a shows a multiple edge $\{u, v\}$, that appears twice. A similar version is shown for

arcs in Example b, in which vertex u is the source of the two arcs $\{u, v\}$, with vertex v as their sink. Example c, however, does not show a multiple arc because the orientation is different: $\{u, v\}$ and $\{v, u\}$. Finally, examples d and e are loops of the form $\{u, u\}$ for edges and arcs.

All graphs considered in this work are graphs without loops or multiple edges. This type of graph is called a *simple* graph.

Let v be a vertex of a graph G. The *degree* of vertex v, denoted by $\delta(v)$, is the number of edges (resp. arcs) connected to v, that is, $\delta(v) = |\Gamma(v)|$. In the directed case, the *indegree* $\delta^-(v)$ is the number of arcs of $A(G)$ of which v is the sink, that is, $\delta^-(v) = |\Gamma^-(v)|$. The *outdegree* $\delta^+(v)$ is the number of arcs of $A(G)$ whose source is v, that is, $\delta^+(v) = |\Gamma^+(v)|$. This notion is extended to the entire graph as follows: the minimal degree of G, $\delta(G)$, is the minimum over all vertices of $V(G)$ of $\delta(v)$:

$$\delta(G) \stackrel{\text{def}}{=} \min\{\delta(v) | v \in V(G)\} \ . \tag{1.4}$$

Similarly, the maximum degree of G, denoted $\Delta(G)$, is the maximum of $\delta(v)$ over all vertices v of $V(G)$:

$$\Delta(G) \stackrel{\text{def}}{=} \max\{\delta(v) | v \in V(G)\} \ . \tag{1.5}$$

A *path* between two vertices u and v of a graph G is a sequence of edges (resp. arcs), $\{\{v_0, v_1\}, \{v_1, v_2\}, ..., \{v_{k-1}, v_k\}\}$ in $E(G)$ (resp. $A(G)$) such that $u = v_0$ and $v = v_k$. The set of vertices in a path p is noted by $V(p)$. The number of vertices in a path p is defined as $|V(p)|$. Let $P(G)$ the set of paths for a graph G. The *length* of a path p is commonly defined as the sum of the weights of the edges (resp. arcs) in p. In the unweighted case, the length correspond to the number of edges (resp. arcs). From the definition of path, we can define a *cycle* as a path in which the start and end vertices are identical.

A graph G is said to be *connected* if there exists a path between any pair of vertices in the graph. In a graph G, a *shortest path* between two vertices u and v is a path of minimal length between u and v. Therefore, the *distance* between u and v is the length of a shortest path between them. From the previous definitions, we define the *diameter* of a connected graph G, denoted $\text{diam}(G)$, by a maximum, for any pair of vertices u and v of $V(G)$, of the distance between u and v. Let u be a vertex of a connected graph G; u is said to be *peripheral* if there exists a vertex v in G such that the distance between u and v is equal to the diameter of G. For more details on the definitions of graphs, please refer to [21].

1.1.2 Hypergraphs

First introduced by Claude Berge [21], hypergraphs (resp., directed hypergraphs) are a generalization of graphs (resp. directed graphs) in which the notion of edge

(resp. arc) is extended to that of hyperedge (resp. hyperarc), which can connect one or more vertices (resp., one or more source vertices to one or more sink vertices). In the general case, a hyperarc can have a source equal to its sink.

Definition 1.1.3. *A hypergraph $H \stackrel{def}{=} (V, E)$ is defined as a set of vertices V, and a set of hyperedges E. A hyperedge is an unordered subset of two or more vertices.*

In this thesis, the sets V and E are considered to be finite. To denote the sets of vertices and hyperedges of a hypergraph H, we can use a similar notation $V(H)$ and $E(H)$ as the ones defined for graphs. We denote by W_V the set of vertex weights, and by W_E the set of hyperedge weights. The weights can be multivalued.

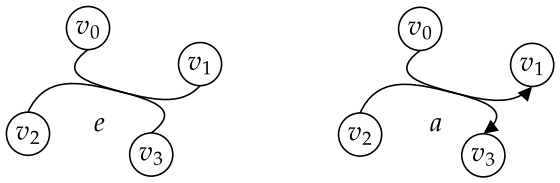


Figure 1.3: An example of hyperedge (e) and hyperarc (a).

Definition 1.1.4. *A directed hypergraph $H \stackrel{def}{=} (V, A)$ is defined as a set of vertices V and a set of hyperarcs A. A hyperarc a is a set composed of two subsets of vertices such that $a = \{s^-, s^+\}$. These sets are $s^-(a) \subset V$ and $s^+(a) \subset V$, with $s^-(a)$ the set of sources of the hyperarc, and $s^+(a)$ the set of its sinks.*

In this thesis, the sets V and A are considered to be finite. $A(H)$ denotes the set of hyperarcs of a directed hypergraph H. We denote by W_V the set of vertex weights, and by W_A the set of hyperarc weights. Weights can be multivalued. Figure 1.3 shows examples of hyperedges and hyperarcs.

Figure 1.4 shows several examples of loops and multiple hyperedges and hyperarcs. Example a shows a multiple hyperedge in which $e_0 = \{v_0, v_1, v_2, v_3\} = e_1 = \{v_0, v_1, v_2, v_3\}$. A similar version is shown for hyperarcs in Example b, in which vertices v_0 and v_2 are the sources of the two hyperarcs $\{\{v_0, v_2\}, \{v_1, v_3\}\}$ with vertices v_1 and v_3 being their sinks. However, in Example c, there is no multiple hyperarc because the orientation is different. The sources $s^-(a_0) = \{v_0, v_2\}$ are different from $s^-(a_1) = \{v_1, v_3\}$. Finally, Examples d and e are loops of the form $\{v_0\}$ for hyperedges and, $\{\{v_0\}, \{v_0\}\}$ for hyperarcs.

As with the graphs defined in the previous subsection, we will not consider hyperloops, multiple hyperedges and multiple hyperarcs.

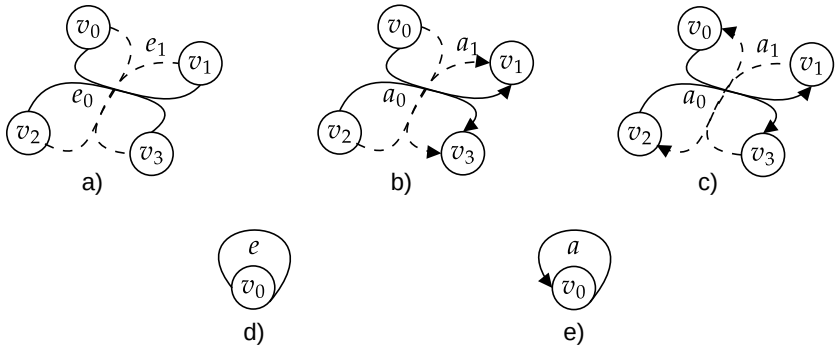


Figure 1.4: Examples of hyperloop and multiple hyperarcs and hyperedges. We dotted hyperedge (resp., hyperarc) to avoid confusion.

The set of neighbors can be extended to the hyperedges and the hyperarcs as follows:

$$\Gamma(u) \stackrel{\text{def}}{=} \{v | \exists e \in E(H) \ s.t. \ u, v \in e, v \neq u\} \ . \tag{1.6}$$

In the case of directed hypergraphs, we have to consider the pairs of source and sink vertices in hyperarcs. Hence, the incoming neighbor set of vertex u is defined as:

$$\Gamma^-(u) \stackrel{\text{def}}{=} \{v | v \neq u, \exists a \in A(H), u \in s^+(a), v \in s^-(a)\} \ . \tag{1.7}$$

Let u be a vertex; the set of its outgoing neighbor vertices $\Gamma^+(u)$ is defined as:

$$\Gamma^+(u) \stackrel{\text{def}}{=} \{v | v \neq u, \exists a \in A(H), v \in s^+(a), u \in s^-(a)\} \ . \tag{1.8}$$

Neighbor relationships of a hypergraph $H = (V, E)$ can be modeled as a graph $G = (V, E')$ whose edges are the results of the transformation of each hyperedge into a clique, also known as clique-net graph in the literature [7]:

$$E' = \bigcup_{e \in E} \bigcup_{u \in e, v \in e, u \neq v} \{u, v\} \ . \tag{1.9}$$

In the directed case, neighbor relationships of a directed hypergraph $H = (V, A)$ can be modeled as a directed graph $G = (V, A')$. The arcs of this graph are the results of the cartesian product of subsets s^- and s^+:

$$A' = \bigcup_{a \in A} s^-(a) \times s^+(a) \ . \tag{1.10}$$

As multiple edges and loops are not considered in this work, we assume that E' and A' do not contain any of them, that is, the graph G that models the neighbor

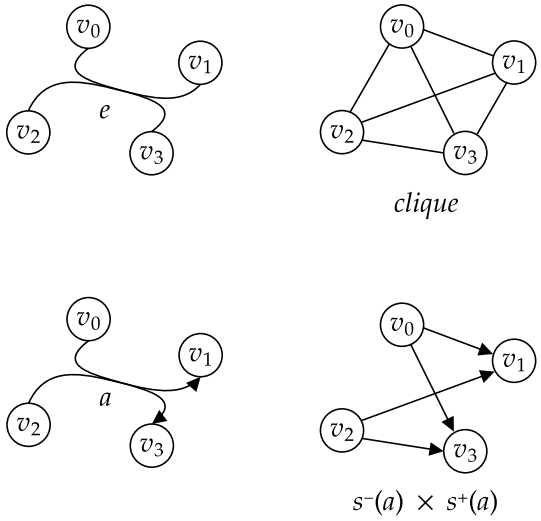


Figure 1.5: An example of a hyperedge and a hyperarc and their respective associated graph representation. Hyperedge e connects all the vertices and the associated subgraph is a clique connecting all vertices in e. In the example below, which concerns a hyperarc a, the cartesian product is obtained between the sources of a and its sinks. We only get the edges that connect the sources with the sinks.

relationship is simple. Figure 1.5 shows an example of a graph model associated with a hyperedge or hyperarc.

This representation is also called the *2-section*, *clique graph* or *primal graph* of some hypergraph H and the directed graph-based representation for directed hypergraph.

Let v be a vertex of some hypergraph H. The *degree* of vertex v, denoted by $\delta(v)$, is the number of hyperedges of $E(H)$ connected to v. This notion is extended to the entire hypergraph as: the *minimal degree* of H, $\delta(H)$, is the minimum, over all vertices of $V(H)$, of $\delta(v)$:

$$\delta(H) \stackrel{\text{def}}{=} \min\{\delta(v)|v \in V(H)\} \ . \tag{1.11}$$

Similarly, the *maximum degree* of H, denoted $\Delta(H)$, is the maximum of $\delta(v)$ over all vertices v of $V(H)$:

$$\Delta(H) \stackrel{\text{def}}{=} \max\{\delta(v)|v \in V(H)\} \ . \tag{1.12}$$

A *hyperpath*, or *path*, between two vertices u and v of a hypergraph H is a sequence of hyperedges $\{e_0, e_1, ..., e_{k-1}, e_k\}$ in $E(H)$, or A(H) in the directed case,

such that $u \in e_0$ and $v \in e_k$ and $\forall i \in \{1, ..., k\}$, $\exists v_i \in e_{i-1} \cap e_i$. In the directed case, u must be a source of e_0, *i.e.*, $u \in s^-(e_0)$ and v must be a sink of e_k, *i.e.*, $v \in s^+(e_k)$ and $\forall i \in \{1, ..., k\}$, $\exists v_i \in s^+(e_{i-1}) \cap s^-(e_i)$. In other words, the path set of a hypergraph is defined by the path set of its 2-section. We denote $P(H)$ the set of paths of some hypergraph H. Let graph G be the 2-section of H; we assume that $P(H) = P(G)$. The set of vertices in a path p is defined by $V(p)$. The number of vertices in a path is defined as $|V(p)|$.

In a directed graph or hypergraph, if a path exists between two vertices u and v, then v is said to be *reachable* from u.

A hypergraph H is said to be *connected* if there exists a path between any pair of vertices in the hypergraph.

A *topological sort* or a *topological order*, is a strict total order of the vertices of a directed graph or hypergraph, such that the index of any vertex v is higher than those of all its in-neighbors. A topological sort of the vertices exists if and only if the directed graph or hypergraph has no cycles, that is, if it is a directed acyclic graph (DAG) or hypergraph (DAH). A topological sort allows one to traverse an acyclic dependency graph or hypergraph so that a vertex is traversed only after all its dependencies have been traversed.

For more details on the definitions related to hypergraphs, please refer to [21].

In the above, have introduced the hypergraph structures that exist in the literature. In this thesis, we are going to use these structures to model circuits, in order to partition them. However, these structures have some limitations, which we are going to explain in more detail in the following subsection.

1.1.3 Red-Black Hypergraph

In this thesis, we introduce the notion of *red-black hypergraphs*. This contribution has been presented in [160]. This structure is specifically designed to model the properties of digital electronic circuits more accurately. Let us recall some of the properties of circuits, presented in the introduction section. A digital circuit is an object consisting of cells interconnected by wires. Hypergraphs seem to be a suitable abstraction for modeling such a group of multi-cell interconnections, with wires connecting two or more cells. Each wire, also called net, propagates a logic state from a driver (source cell) to receiver cells. This property can be modeled by specific hyperarcs, in which the number of sources (vertices) is limited to one for each hyperarc. Each logic state is emitted by a synchronous cell called a *register*. A clock signal is used to synchronize the operations of all logic parts of the circuit, that are performed simultaneously. A *clock signal* is a logic signal that periodically activates the register cells at a constant global frequency. The *frequency* is the number of clock events, *i.e.*, clock ticks, per second. Synchronicity ensures that a result calculated by one part of the circuit is available at the end

of the clock period for use as an input value in another part of the circuit for the next clock period.

Another main type of cells is the *combinatorial cells*. Each combinatorial cell performs a logical operation from its input logic states and produces one or more logic states as output. Examples of combinatorial cells can vary from a simple logic gate to a complex arithmetic compute block. Each combinatorial cell has a processing time associated with it, which depends on the operations performed. The difference between these two types of cells is that a register updates its state synchronously at each clock tick, and a combinatorial cell propagates a logic state immediately after that logic state has been received and processed.

Some works refer to a netlist to model a digital electronic circuit. A *netlist* is a list of nets (or wires) and cells, with possible additional data of the represented digital electronic circuit. As a model of digital electronic circuit, a netlist can also be represented by a hypergraph. Consequently, we will also refer to digital electronic circuit, netlist, and hypergraph to denote the same object in the sense of a list of vertices connected by hyperedges.

Red-black hypergraphs are a subclass of directed hypergraphs in which each hyperarc contains exactly one source vertex, *i.e.*, $|s^-(a)| = 1, \forall a \in A$, so that each signal has one source cell. Moreover, each vertex is assigned a color, red or black. Red vertices (resp., black vertices) form a subset of vertex set V, $V^R \subset V$ (resp., $V^B \subset V$), such that $V^R \cap V^B = \emptyset$. Red vertices model the register cells, and black vertices model the combinatorial cells. By definition, for $u \in V(H)$:

- if $\Gamma^+(u) = \emptyset$, then $u \in V^R$. u is a global input (source) of the circuit;
- if $\Gamma^-(u) = \emptyset$, then $u \in V^R$. u is a global output (sink) of the circuit.

These conditions are necessary but not sufficient, *i.e.*, a vertex can be red even if it does not satisfy the two conditions defined above.

As hypergraphs can be multivalued, red-black hypergraph vertices can also be multivalued. Let $W_V(u)$ be a weight vector associated with vertex u. In the unweighted case, $W_V(u) = \{1\}$. Multivalued vertex weights can be used to model the multi-resource aspect of the circuits and of the FPGA components.

Figure 1.6 shows an example of a red-black hypergraph modeling a combinatorial circuit. A combinatorial block is a sub-circuit bounded by the cells that are registers or circuit inputs and outputs. Figure 1.7 shows an example of a circuit with two combinatorial blocks. In this example, registers stabilize and forward a logic state. One of the properties of the circuits is that all registers output values are modified simultaneously on each clock tick. The time that elapses between two clock ticks is called the *clock delay*. The processing time of a combinatorial path is defined as the sum of the processing times of the cells belonging to the path. The longest path in a block is called a *critical path*. The critical paths within the

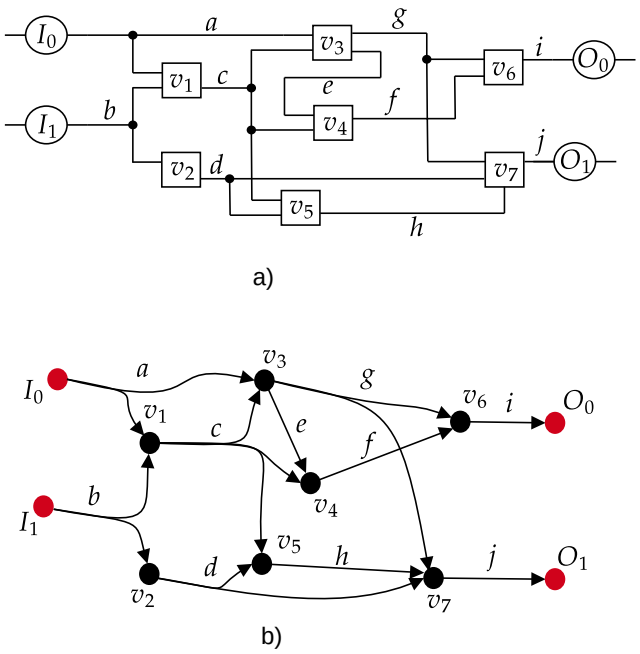


Figure 1.6: The circuit in a) consists of two inputs, two outputs, and 7 combinatorial cells. The red-black hypergraph b) models the circuit above. It consists of 4 red vertices corresponding to the sources and sinks of the circuit, and 7 black vertices corresponding to the combinatorial cells.

combinatorial blocks constrain the clock delay because the computed logic state values must stabilize at the output registers of each combinatorial block before the next clock tick. If this constraint is not satisfied, the circuit will behave in an unexpected manner.

In this thesis, we focus on minimizing the length of the critical path of combinatorial circuits. Since we are only interested in minimizing the propagation times of combinatorial logic between two registers, it makes no sense to consider the length of paths that comprise any additional register different from the start and end registers. The red-black hypergraph model enables the extraction of all combinatorial paths, by considering only paths whose start and end vertices are red.

Since we are working exclusively on synchronous circuits that are designed for implementation on FPGA, we are guaranteed that there are no combinatorial loops, *i.e.*, infinite paths that are made of an infinite repeated pattern of combinatorial cells. That is, a cycle in a red-black hypergraph contains at least one red vertex. As a result, a combinatorial block, such as the one shown in Figure 1.6,

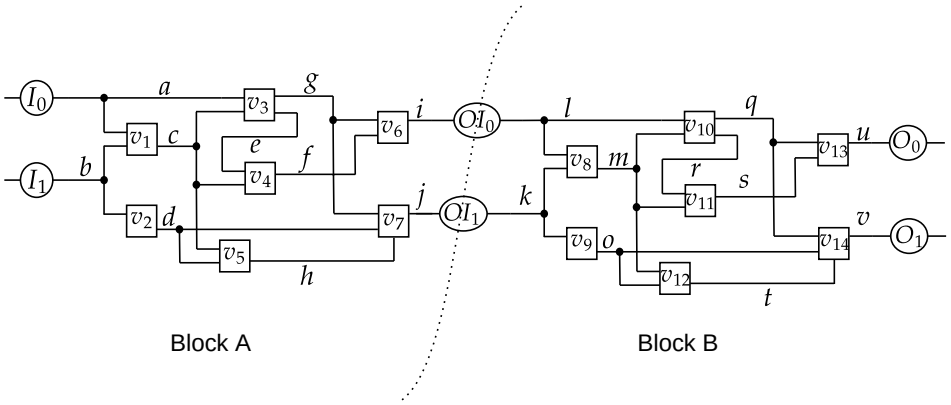


Figure 1.7: Circuit composed of two combinatorial blocks. The OI_x cells are both outputs of block A and inputs of block B.

forms a red-black directed acyclic hypergraph (DAH). Since, a circuit is made up of several such combinatorial blocks, a red-black hypergraph is composed of multiple DAHs interconnected by their red vertices. Finally, the paths to be considered within the red-black hypergraph are *red-red* paths that are the set of all paths of each DAH, starting from a red vertex and ending to a red vertex. A red-red path p for a DAH h is defined as:

$$p = \{(v_0, v_1), ..., (v_{n-2}, v_{n-1})\} \ , \tag{1.13}$$

with $v_0, v_{n-1} \in V^R(h)$ and $v_i \in V^B(h), i \in \{1, ..., n-2\}$. Let $H = \{h_0, ..., h_{k-1}\}$ be a red-black hypergraph composed of k DAHs; the set of red-red paths $P^R(H)$ is defined as:

$$P^R(H) = \bigcup_{h_i \in H} P^R(h_i) \ . \tag{1.14}$$

If not specified, we will assume that H defines a red-black hypergraph and H a directed acyclic hypergraph.

Note that $P^R(H)$ is a subset of the paths in H. Hence, $P^R(H) \subset P(H)$. Each path has a length, or cost, which models the distance between two vertices. Classically, the length of a path equals the number of arcs along the path, *i.e.*, the number of vertices in p minus one for the unweighted case. In a digital electronic circuit, wires and cells have a delay associated with them. In this case, vertices and hyperarcs should be weighted: the length of a path is the sum of the weights of the arcs. The arc weights are defined by $d(u, v)$, and the weights associated with the vertices, by $d(u)$. The length, or traversal cost, d, can be extended to a path p as follows:

$$d(p) = \sum_{u \in V(p)} d(u) + \sum_{u \in s^-(a), v \in s^+(a), a \in p} d(u,v) \ . \tag{1.15}$$

In the remainder of this document, in the absence of precision, for any path p, we will use the terms *path length* or *path cost* to refer to the quantity $d(p)$.

By extension, $d_{\max}(u,v)$ can be defined as the value of the longest path between vertices u and v:

$$d_{\max}(u,v) = \max\{d(p), p \in P_{u,v}\} \ , \tag{1.16}$$

were $P_{u,v}$ is the set of paths between u and v. Furthermore, in circuit prototyping on a multi-FPGA platform, the routing process is generally performed after the partitioning. This thesis only focuses on the partitioning step. As a result, there is no specific cost associated with connected pair of vertices because, at this step, there is no routing cost (arc weight). In the absence of precision, we will assume that the weight of each arc is zero when calculating the length of a path $d(p)$.

Using the above definitions, we can define the critical path of a red-black hypergraph H as:

$$d_{\max}(H) = \max\{d(p) | p \in P^R(H)\} \ . \tag{1.17}$$

Note that, with a topological sorting of DAH in H, it is possible to compute $d_{\max}(H)$ in polynomial time.

1.2 Criticality

In this section, the concept of criticality is defined. This notion is the basis of several circuit partitioning methods. Criticality is used to model the timing associated with the cells in a circuit.

1.2.1 Circuit cell criticality

Criticality is a metric used in [4, 35] and our work [160], to classify the cells of a circuit according to the cost of the combinatorial path they traverse. The *criticality* of a vertex v is equal to the length of the longest path traversing v.

Figure 1.8 shows an example of a critical path in a circuit. Note that the paths considered are only combinatorial, i.e., paths whose start and end are a register.

Different weighting schemes and methods to compute the criticality are detailed in Chapter 4. One such method is to compute the criticality for each vertex by calculating the cost of the longest path traversing it. The longest path can be estimated by first, propagating the accumulated delays across each DAH, from the sources to all other vertices. At the end of the propagation, each sink register at

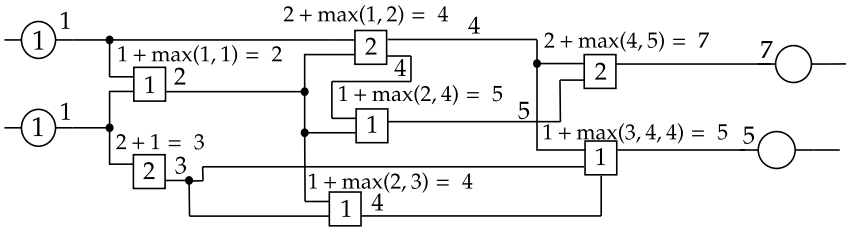


Figure 1.8: In this example, delays are propagated from source registers to sink registers. Only locally maximal value are propagated along paths. This yields a critical path equal to 7 for the circuit.

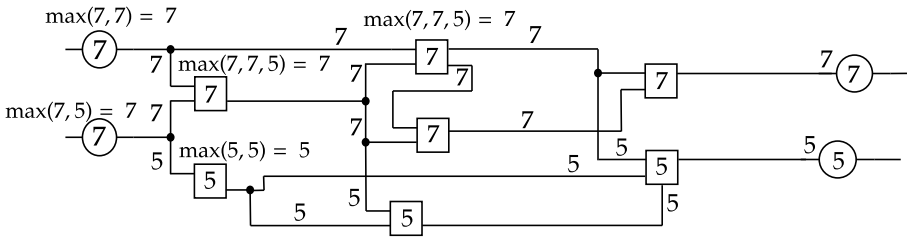


Figure 1.9: In this example, critical path lengths are computed by back-propagation from the sink registers to the source registers. Only maximum critical value are back-propagated. Then, criticalities for each vertex is equal to the length of the critical path (7) and the length of the second path (5).

the end of a path is assigned a total propagation time equal to the longest path between that register and any of its predecessor vertices belonging to the path. If we back-propagate the maximum path value from the sinks back to the source registers, we obtain for each vertex a criticality value that is an upper bound on the value of the longest path traversing it. Figure 1.9 shows an example of critical path value back-propagation. This process labels cells with a value defined as the cell's criticality.

The criticality of a cell is equal to the upper bound of the longest path back-propagated from the output/sink registers.

1.2.2 Vertex and path criticality

The notion of criticality can be extended to the red-black hypergraph. The procedure for calculating critical paths by delay propagation from source to sink registers is identical, as they are identified as red vertices. Since there are no combinato-

rial loops within circuits, red-black hypergraphs have this property, specifically for any directed acyclic sub-hypergraph. As a result, it is possible to propagate delays associated with vertices in polynomial time, e.g., using topological sorting, in any DAH. The back-propagation is performed in reverse topological order.

1.3 Partitioning and Clustering

In this section, a vertex set partition, or equivalently a set of clusters, *i.e.*, a subset of vertices, is defined. A *partition* is a set of subsets of vertices, disjoint or not, depending on the nature of the problem. Several objective functions are available for different applications, such as static process placement [151] or domain decomposition. In this thesis, we focus on circuit partitioning for rapid prototyping.

1.3.1 Partitions and cut metrics

A *k-partition* or a *k-way* partition Π of $H = (V, A)$ is the splitting of V into k vertex subsets $\pi_1..\pi_k$, called *parts*, such that:

(i) all parts π_i, given a capacity bound M_i on vertex weights, respect the capacity constraint: $\sum_{v\in\pi_i} W_V(v) \leq M_i$;

(ii) all parts are pairwise disjoint: $\forall i \neq j, \pi_i \cap \pi_j = \emptyset$;

(iii) the union of all parts is equal to V: $\bigcup_i \pi_i = V$.

A *bipartition* is a partition of vertices composed of two parts. For a given partition Π of H, the *connectivity* $\lambda_\Pi(a)$ of some hyperarc $a \in A$ is the number of parts connected by a. If $\lambda_\Pi(a) > 1$, then a is said to be *cut*; otherwise, it is entirely contained in a single part and is not cut [155, 55]. The *cut* of some partition Π is the set $\omega(\Pi)$ of cut hyperarcs, defined as:

$$\omega(\Pi) \stackrel{\text{def}}{=} \{a \in A, \lambda_\Pi(a) > 1\} \ . \tag{1.18}$$

It is also possible to define the set of vertices that belong to the cut hyperarcs. This subset of vertices is commonly referred to as the *frontier*, or *halo*:

$$\hbar(\Pi) \stackrel{\text{def}}{=} \{v | \exists a \in \omega(\Pi), v \in a\} \ . \tag{1.19}$$

One possible metric to evaluate a partition is to sum the weight of cut hyperarcs. If at least two parts share some hyperarc, then this hyperarc is cut. The cut size f_c is defined as:

$$f_c \stackrel{\text{def}}{=} \sum_{a\in\omega(\Pi)} W_A(a) \ , \tag{1.20}$$

with $W_A(a)$ the weight of hyperarc a. If all hyperarcs have the same weight (equal to 1), the cut size is equal to $|\omega(\Pi)|$. Another metric used in some partitioning problems to measure the quality of partitions is called *connectivity-minus-one*. The connectivity-minus-one cost function f_λ of some partition Π of a hypergraph H is defined as:

$$f_\lambda \stackrel{\text{def}}{=} \sum_{a \in A} (\lambda_\Pi(a) - 1) \ . \tag{1.21}$$

Figure 1.10 shows an example of a partition and its cut and connectivity costs.

In contrast to the min-cut function f_c, which counts the cut hyperedges, the connectivity cost function counts the number of parts in which the vertices of each hyper-edge are located. This makes it possible to model the communication cost for a task-to-process mapping problem.

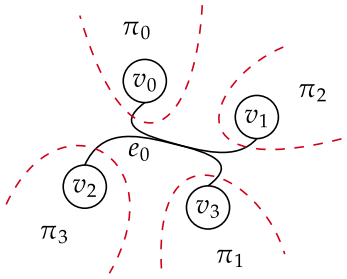


Figure 1.10: In this example, each vertex is placed in a different part. The hyperedge e_0 is therefore considered to be cut. The size of the cut, f_c, is 1. The connectivity-minus-one cost, f_λ, which measures the number of shared parts for each hyperedge, is 3, because there are 4 different parts connected by e_0.

1.3.2 Problem statement

This subsection defines the problem addressed in this thesis. The problem of partitioning red-black hypergraphs, considering the path length and a target topology, consists in finding a partition of the vertex set V of H that minimizes the degradation of the length of the critical path.

A partition Π degrades the path length by adding a cut penalty for each cut hyperarcs along that path. The cut penalty model embeds a notion of *distance* that represents the cost of signal propagation between two distant elements on the target platform. This makes it possible to roughly take into account routing times on a multi-FPGA platform for which not all pairs of FPGAs would be directly connected. Figure 1.11 shows an example of a cut path. The cut cost may depend on the involved pairs of parts. Based on Figure 1.12 example, we can consider for

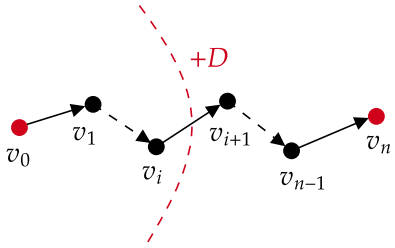


Figure 1.11: In this example, the path is cut between vertices v_i and v_{i+1}. The length of this path is therefore increased by D, associated with the cut.

a 4-partition, the cut cost between parts π_0 and π_1 is 10, and the cost between π_0 and π_2 is 200. In this specific case, avoiding spreading a critical path between parts π_0 and π_2 may make more sense. This heterogeneity is most often due to the fact that the topology is not fully connected, so that additional FPGAs have to be traversed for certain routes.

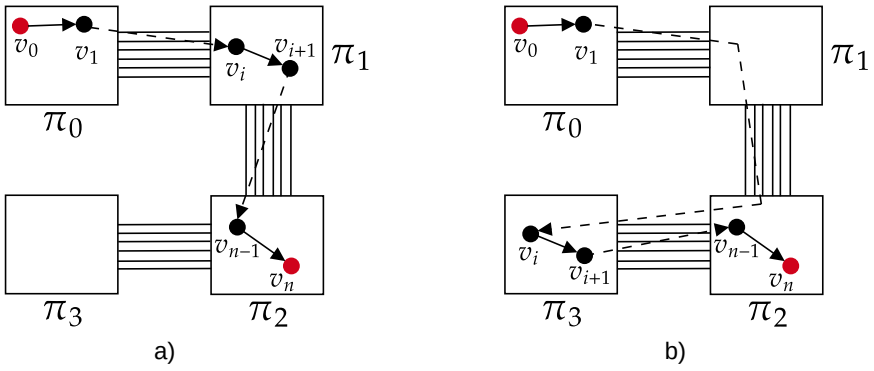


Figure 1.12: Example of paths that are placed on a not fully connected target topology. Placement a) generates two routing (cut) costs, while placement b) generates two cut costs plus two additional routing costs. Placement a) is therefore less costly than b).

In these examples, individual routing costs must be evaluated if critical path degradation is to be minimized. It is therefore possible to extend the definition of the path length for a path p on a red-black hypergraph $H = (V, A)$ according to a partition of its vertices Π, as follows:

$$d^{\Pi}(p) = d(p) + \sum_{u \in s^-(a), v \in s^+(a), a \in p} D_{\pi(u),\pi(v)} \quad , \tag{1.22}$$

with $\pi(u)$ being the part containing u and $D_{\pi(u),\pi(v)}$ the cut penalty between part $\pi(u)$ and part $\pi(v)$. If u and v belong to the same part, the cut cost $D_{\pi(u),\pi(v)}$ is equal to zero.

The problem of partitioning red-black hypergraphs, considering the path length and the target topology, consists in finding a partition of the vertex set $H(V)$ that minimizes also the objective function f_p, and that respects the capacities of each part. Let H be a red-black hypergraph and Π a partition of its vertex set; the function f_p is defined by:

$$f_p \stackrel{\text{def}}{=} \max\{d^{\Pi}(p) | p \in P^R(H)\} \ . \tag{1.23}$$

In this thesis, we will focus on the function f_p, which is the main objective of our work. Since the number of connections between parts is limited due to physical constraints, the cut size does have to be taken into account. However, more than the cut size metric is needed to correctly model the effects of multiplexing on the final delay. Signal multiplexing is a technique allowing to transfer a set of signals between two logic elements, even if the transfer capacity per cycle is limited. This technique results in subsequent delays, to have time to transfer all the signals. For example, consider a capacity c between two parts. An extra delay must be used if the number of signals between the two parts exceeds the capacity c. If an extra delay is used, the capacity is no longer c, but at least $2c$. Hence, a cut size of $2c$ is as good as $c+1$, while from the point of view of the functions f_c and f_λ, it is not.

1.3.3 Graph and hypergraph circuit models

In the previous sections, we presented some limitations of graphs and hypergraphs for representing certain properties of a digital electronic circuit. These properties are important for the partitioning problem addressed in this thesis. In addition, this subsection introduces representations, and their limitations, for modeling a digital electronic circuit[1]. Some works [8, 37, 102, 125] introduce their circuit model and associated approach to partition a circuit. There are several ways to model directed hypergraphs as graphs; H. R. Charney *et al.* [37] present a transformation of each hyperedge into a clique. This model is called the *clique model* because an edge is created for each pair of vertices in each hyperarc a. For a directed hypergraph $H = (V, A)$, the graph associated with H using the *clique model* is the graph $G = (V, E)$ such that $E = \{\{u, v\} | \exists a \in A, u, v \in a\}$ [37]. The consequence of this model is to create many edges. In the clique model, several methods of edge weighting are used to best reflect the hyperedge cut cost. The goal is to get a similar cost for the same partition of vertices in H and in G. For

[1] Readers interested in other, more general, models, particularly for non-directed hypergraphs, may consult [8].

example, if a hyperedge e is cut in H, then the set of edges that model e in G is cut, and the total cost for cutting them should be 1. However, E. Ihler *et al.* [102] prove that such a "perfect" clique network model is impossible. Furthermore, T. Lengauer [125] shows an $O(\sqrt{|e|})$ deviation for bi-partitioning the hyperedge e, regardless of the cost function used.

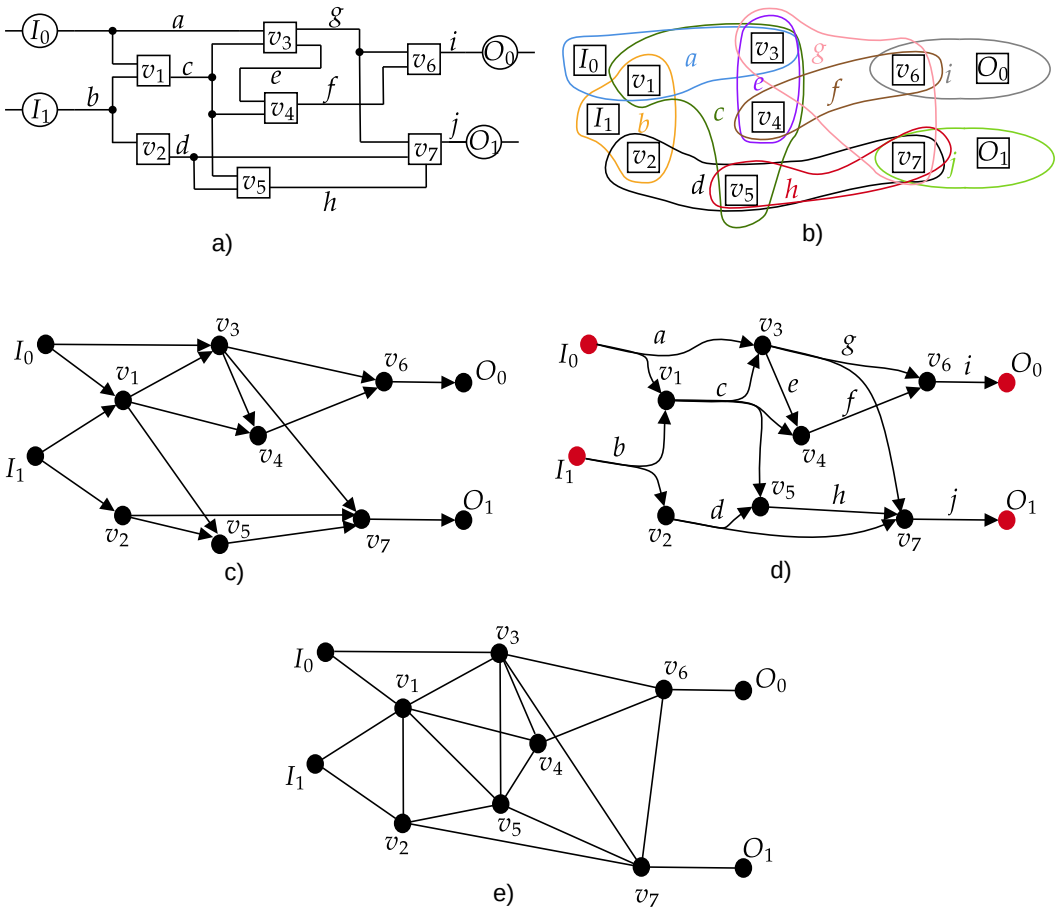


Figure 1.13: Representations of a circuit with 7 modules, 2 inputs, 2 outputs and 10 signal nets (or wires): (a) circuit diagram with all modules' inputs on their left side and all their outputs on their right side; (b) the associated hypergraph representation; (c) the directed graph, assuming a directed-tree hyperedge model; (d) the red-black hypergraph model, in which nets are modeled as hyperarcs, registers by red vertices, and combinatorial cells by black vertices; (e) the clique model.

In the case of the directed graph model [8], only edges between the source

of the hyperedge and its sinks are created. The directed graph model allows one to represent the relationships between the components of a circuit as well as the combinatorial paths associated with the circuit. For a directed hypergraph $H = (V, A)$, the graph associated with H using the directed graph model will be the graph $G = (V, E)$ such that: $E = \{\{u, v\} | \exists a \in A, u \in s^-(a), v \in s^+(a), a \in A\}$. Figure 1.13 shows an example hypergraph and the corresponding clique and directed graphs.

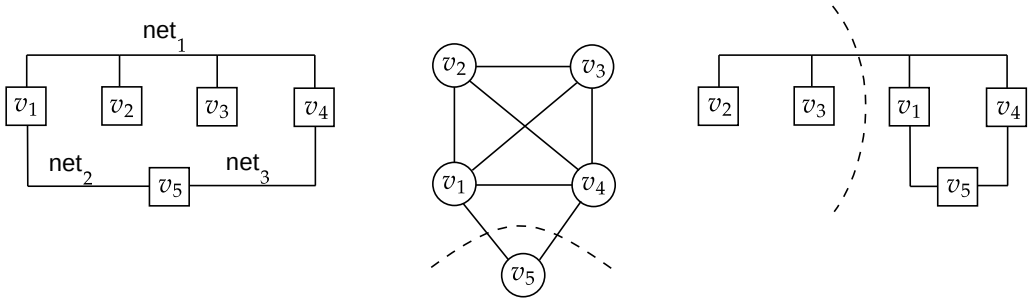


Figure 1.14: An example of the limitations of the graph-based model for partitioning a circuit to minimize cut size. In this example, (a) the circuit is composed of 5 cells connected by 3 wires; (b) in this specific case, an optimal graph-based partition would place cell v_5 in one part and all the other cells in the other, as this is the only two-part partition that results in only two cut edges; (c) however, by placing cells v_2 and v_3 of the circuit in one part and cells v_1, v_4, and v_5 in the other, we obtain a partition in two parts, which requires only a single wire crossing the cut. The example is taken from [176].

Models should be adapted to represents all the properties needed by a objective function for partitioning. One of the objective of circuit partitioning is to partition a circuit into two or more parts, while minimizing the number of wires connecting the circuit elements between different parts. In practice, minimizing the number of wires is crucial because it reduces the wiring cost and the configuration's total area [192, 8]. In the context of circuit partitioning in Very Large Scale Integration (VLSI) design, R. A. Rutman [164] and D. G. Schweikert and B. W. Kernighan [176] have shown that graphs are a limited representation of a circuit. Figure 1.14 shows an example of the limitations of the graph-based model for partitioning a circuit to minimize the cut size. In graph modeling, each cell of the circuit corresponds to a vertex, and a wire connecting several circuit elements is represented by adding a clique of edges connecting all pairs of connected vertices. A significant difference between the circuit partitioning solution and the graph

partition arises because graphs cannot accurately represent relationships between more than two objects. That is, if a hyperedge is cut, many edges would be cut in the graph-based partition. As a result, a hypergraph-based partition is more accurate in modeling the partitioning of a circuit while minimizing the size of the cut.

In addition, the plain hypergraph model does not contain any information about the combinatorial paths. Consequently, hypergraphs must be extended to contain properties that are relevant to the problem addressed in this thesis. Thus, the red-black hypergraphs introduced in this thesis are specifically designed to model the properties of digital electronic circuits that are missing in hypergraphs, for partitioning circuits with the cost function f_p.

1.4 Conclusion

In this chapter, the notations and definitions used in this thesis have been introduced. Table 1.1 summarizes the most important notations defined above, which will be used throughout the document. However, not all notations are listed here, only a summary of the most important ones.

We have introduced the new concept of red-black hypergraphs, which are an extension of hypergraphs. This enriched model is necessary to consider all the relevant properties associated with digital electronic circuits, to provide metrics and data that are closer to reality. The ability to calculate the length of a path and its degradation during a partitioning operation is an essential issue for minimizing the degradation of the critical path. We have shown that previous state-of-the-art circuit representations cannot be used to model such paths adequately.

Table 1.1: Notation summary

Variable	Definition
$G = (V, E)$	Graph
$H = (V, A)$	Red-black directed hypergraph
V^R	Set of red vertices
V^B	Set of black vertices
v	Some vertex $v \in V$
a	Some hyperarc $a \subset V$
$\Gamma(v)$	Vertex set of neighbors of v
$\Delta(H)$	The maximum vertex degree in H
Π	Partition, with $\Pi[v]$ the part of v
π_i	Part i, with $\pi_i[v] = 1$ iff $v \in \pi_i$ and 0 else
M_i	Capacity of part i
p	Some path $p \in P$
$d_{\max}(u, v)$	Maximum distance between u and v
$d_{\max}(H)$	Longest path distance for H
$d^{\Pi}_{\max}(H)$	Longest path distance for a partitioned H^{Π}
D_{ij}	The path cost between parts i and j
$\omega(\Pi)$	Cut of a partition Π
$\hbar(\Pi)$	Halo of a partition Π
λ_a	Connectivity of hyperarc $a \in A$
$\Lambda(H)$	The maximum connectivity in H
λ_v	Connectivity of vertex $v \in V$
$\Lambda_V(H)$	The maximum connectivity of vertices in H
f_λ	Connectivity-minus-one cost function
f_c	Cut cost function
f_p	Longest-path cost function

Chapter 2

State of the art in circuit and hypergraph partitioning

Circuit partitioning is helpful in some main practical applications for VLSI design as circuit prototyping or circuit layout. In the case of circuit prototyping, circuits are partitioned so as to map them onto elements of the FPGA, in order to not exceed their capacities and not to degrade the critical paths. In the case of circuit layout, circuits are partitioned so as to map them onto a 2D layout *i.e.*, on the silicon surface of the integrated circuit (IC), in order not to degrade the critical paths. However, differences exists in the weighting and evaluation of path criticality. For circuit placement on multi-FPGA platforms, a synthesis step is mandatory, and the target FPGA technology constrains each cell's time. In addition, the target platform may not be fully connected, creating additional routing constraints. 2D placement imposes different constraints.

Each of the works on critical path modeling has been of interest. Even if the objective is not for placement on a multi-FPGA platform, the problem of partitioning a circuit while minimizing the impact on the critical path is an approach worth studying. The subject of this thesis needs to be placement within the FPGA. As a result, the subject is sufficiently high-level to take advantage of work relating to circuit partitioning with path minimization while maintaining consistency with the practical problem. This objective remains essential for both problems.

This chapter presents a broad overview of the state-of-the-art related to the graph and hypergraph partitioning problem for VLSI design. Some of these reference works deal with minimizing the cut size, minimizing the critical path degradation, or both. A more detailed explanation of some of these works can be found in the dedicated sections.

2.1 Partitioning methods and applications

This section presents different approaches from direct hypergraph partitioning. Due to the size of the instance, it is sometimes useful to reduce the hypergraph and partition its reduced version, especially to reduce complexity and computation time. For example, recursive bipartitioning is very effective at reducing computation time.

2.1.1 Hypergraph bipartitioning

There are two main approaches to computing a k-way partition of a hypergraph. The first one is to perform a recursive bipartition (RB), *i.e.*, to first compute a bipartition of the initial hypergraph, then by recurring on each of the two parts, we end up with a k-partition. To obtain k parts, $O(k)$ bipartitioning steps are required to partition the hypergraph. The bipartitions computed in the RB approach form a binary tree called a bipartition tree [180]. In k-way direct partitioning, the

hypergraph is directly partitioned into k parts without going through the 2-way recursive approach. Note that there are strategies that combine both, RB and k-way partitioning.

2.1.2 Graph-based hypergraph partitioning

Several works have addressed the problem of partitioning hypergraphs by using graph partitioning algorithms, via graph models of hypergraphs. However, E. Ilhler *et al.* [102] showed that it is not possible to model a hypergraph H as a graph G such that, for any bipartitioning of the vertex set, the result for the functions f_λ and f_c of the cut in H is the same as for G. The result remains similar even with the addition of dummy vertices, unless negative weights on edges are allowed. Regarding the clique model presented by T. Lengauer in [125], it is shown that there is always a bipartition with a deviation of $\Omega(\sqrt{|e|})$ from the desired unit cost, regardless of the weighting scheme.

2.1.3 Static mapping

Static graph mapping is the problem of finding a mapping between two graphs: the source graph G_s and the target graph G_t. The number of vertices in G_t defines the number of parts with a possible bounded capacity for each part depending on G_t vertices. A practical application of the static mapping problem is the mapping of the processes of a parallel program onto a parallel machine. One of objective for the static mapping problem is to minimize the global communication bandwidth. M. R. Garey *et al.* [78] have shown that static mapping is an NP-complete problem in the general case. F. Pellegrini [152, 148] proposed an algorithm based on the recursive bipartition of the graph G_s and G_t. All his algorithms are implemented in the SCOTCH [149] tool. The reader can refer to several papers dealing with static graph mapping [24, 41, 5, 97, 150, 116, 54, 22].

Mapping is an important notion for our study, since the target topology can be a factor affecting the critical path. Modeling static mapping from the source graph onto a target graph will help the algorithm assign vertices to parts in a way that provides lower routing costs. In this thesis, we are not working on detailed placement/routing on the FPGA, but on the partitioning step that precedes it. However, a coarse-grained, platform-scale placement may allow more optimal pre-placement of vertices in parts, with respect to algorithms agnostic to the target topology.

2.2 Computational complexity

A circuit can be modeled as a graph or hypergaph, that is, graph and hypergraph partitioning is commonly used to obtain a partition of a circuit. Hence, the complexity of circuit partitioning is close to the complexity of graph and hypergraph partitioning. There exist several ways to measure the quality of a partition. Multiple cost functions have been defined in the state-of-the-art, such as f_c or f_λ. Additional constraints can complete costs functions for the partitioning problem, such as capacity limits for each part. During a partitioning process, both cost function and constraints have to be evaluated, that is, the complexity of the process depends on the number of constraints and the complexity of the cost function. This section will look at different complexity results for the graph and hypergraph partitioning problem.

2.2.1 Complexity of partitioning

A partition of a graph creates a cocycle, *i.e.*, the set of edges (resp., hyperedges) across multiple parts. The cocycle of a partition defines a cut, and its size define the cut size. However, finding a minimal cut in a graph is equivalent to finding a minimal (s-t)-cut in G. An *s-t-cut* in a graph G is a minimum edge separator of a source vertex s and terminal vertex t in G. Note that an s-t-cut does not guarantee a balanced number of vertices on both sides of the cut. Using Ford and Fulkerson's max-flow/min-cut theorem [77], we can solve this problem in polynomial time. Some algorithms with an improved time complexity have been proposed to find a minimum cut in a graph and multi-graphs [138, 90]. E. L. Lawler [121] and J. Li *et al.* [126] proposed to compute an (s-t)-flow for hypergraphs to compute the minimum cut. In the case of hypergraph partitioning, the hypergraph can be transformed into a polynomial larger graph. When $k > 2$, the problem becomes NP-hard if k is part of the input [83]. However, if k is part of the algorithm's input data, there is an algorithm with complexity $O(n^{k^2}T(n,m))$ [83], where $T(n,m)$ is the computation time of an (s-t)-cut.

K. Fox *et al.* found an algorithm to compute the minimum k-cut of a hypergraph with complexity $O(mn^{2k-2})$ for any rank and an algorithm with complexity $O(n^{\max\{r,2k-2\}})$ for a fixed rank.

In the case of balanced partitioning, the problem is NP-hard [78, 125] for both graphs and hypergraphs, and there is no constant approximation factor [11], unless $P = NP$.

2.2.2 Complexity of clustering

Z. Donovan *et al.* [60, 61] addressed combinatorial circuit clustering problems when cell replication is allowed (CA) and when cell replication not allowed (CN). In the rest of this dissertation, we use CA and CN to refer to these clustering problems. The objective function is to minimize the degradation of the critical path induced by edge cuts. The problem is NP-hard and has no constant approximation factor. The version of the problem where the number of clusters is fixed with a balancing constraint is at least as hard as clustering. The problem of balanced partitioning of a red-black hypergraph with minimization of critical path degradation and cutting is at least as hard as these previous problems. However, A. A. Diwan *et al.* [57] presented a polynomial algorithm for trees. Z. Donovan *et al.* [60] have also presented an approximation algorithm parameterized by the cluster size M, with the approximation ratio $M^2 + M$. In Chapter 4, we introduce a polynomial algorithm with the approximation ratio parameterized in M under restricted hypothesis.

2.3 Partitioning and mapping tools

Over the last 30 years, several graph and hypergraph partitioning software tools have been developed and made publicly available.

2.3.1 Metis, hMetis and khMetis

G. Karypis *et al.* [106, 109] introduced the tool Metis for partitioning graphs, with an objective function that minimizes the cut size, f_c. Due to the more generalized aspect of hypergraphs, hypergraph partitioning has become a necessity in the scientific community. George Karypis and Vipin Kumar created a hypergraph version, based on recursive bipartitioning, called hMetis [111]. However, as hMetis showed difficulty meeting the part-balancing constraint, the authors introduced a new tool for k-partitioning, called khMetis [112]. The tool authors claim that for many circuits, particularly those with cell counts greater than 100k and non-uniform cell weighting, hMetis produces bisections that cut 10% to 300% fewer hyperedges than those cut by other popular algorithms such as PARABOLI [158], PROP [66], and CLIP-PROP [67].

2.3.2 PaToH and kPaToH

Ümit V. Çatalyürek, and Cevdet Aykanat [34] then released the PaToH partitioning tool. This tool allows the partitioning of multi-constrained hypergraphs, by minimizing the connectivity between the different parts, f_λ. Partitioning for VLSI

design or prototyping on multi-FPGA platforms requires consideration of multiple resources due to the different resources available on an FPGA, such as LUTs, RAM, DSPs, and so on. These resources are constrained within each FPGA (part). Since PaToH allows the definition of several resource capacity constraints during the partitioning process, it seems appropriate for these cases. The authors also introduced a partitioning strategy based on vertex replication, referenced in [178] and an extension of PaToH for k-partitioning, called kPaToH [16] .

2.3.3 KaSPar, KaHIP and KaHyPar

More recently, in 2010, V. Osipov *et al.* [141] released the graph partitioning tool KaSPar, followed by the partitioning framework KaHIP in 2013, by P. Senders *et al.* [169]. S. Shlag [173] presented in his thesis the hypergraph partitioning framework KaHyPar, which embeds several partitioning algorithms. These tools focus more on partition quality than on computation time. KaSPar, KaHIP, and KaHyPar achieve better results in minimizing the cut size than other partitioning tools [85, 10, 94, 93, 95, 6, 174, 175].

2.3.4 TopoPart and TritonPart

D. Zheng *et al.* [193] introduce TopoPart, a topology driven hypergraph partitioner designed for targeting multi-FPGA platform. Like modern hypergraph partitioners, TopoPart uses a multilevel framework to tackle large circuits. TopoPart aim at partitioning a circuit while minimizing the distance between vertices within the topology.

TritonPart is a new hypergraph partitioner introduced in November 2023 by I. Bustany *et al.* in [31]. TritonPart incorporates multiple costs, e.g., cut size, timing, and embedding. TritonPart is also based on a multilevel framework.

2.3.5 SCOTCH and PT-SCOTCH

Graph static mapping consist of map a source graph G_s onto a target graph G_t. In G_t, edges represent connections, and weights on edges represent, for example, transmission capacity or traversal cost. In 1994, F. Pellegrini introduced an algorithm [152] based on dual recursive bipartitioning for static mapping problem and released in 1996, SCOTCH [151], a graph partitioning and mapping tool that considers a target topology. The function to be minimized is different from that of the previous tools. The goal of SCOTCH is to place graph G_s on a target graph G_t so that the expansion of the edges between the vertices of G_s placed on G_t is minimized and the placement of the vertices is balanced. A parallel version of SCOTCH exists since 1996, under the name PT-SCOTCH [40].

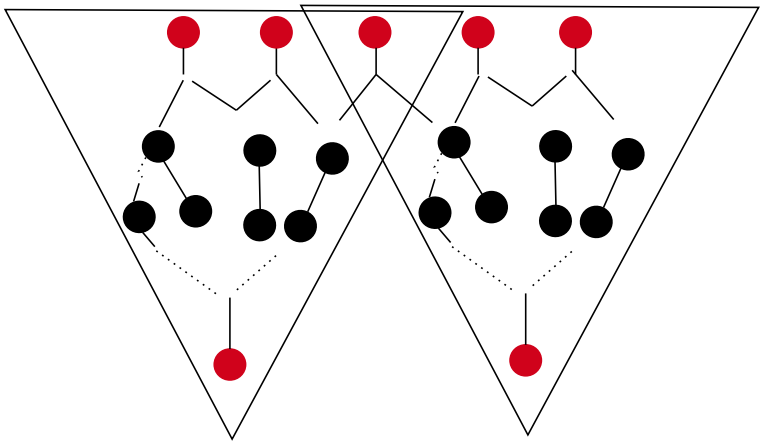


Figure 2.1: An example of a red-black hypergraph with two cones.

2.3.6 Conclusion

In this section, we presented some popular partitioning tools. We presented only the most common and publicly available tools probably used in circuit partitioning. However, each tool is designed for a specific cost function, such as f_c, f_λ, and so on. There are practical problems for which it is necessary to specialize the cost function. This is the case of the problem treated in this thesis with the function f_p.

2.4 Algorithmic approaches to hypergraph partitioning and placement

This section presents the various approaches that have been developed to partition circuits for different practical purposes, such as circuit placement on a 2D surface with critical path minimization, or circuit prototyping on a multi-FPGA platform. Some approaches use the tools presented in the previous section in conjunction with pre- and post-processing to drive the tool with practical constraints, such as cell spacing for 2D placement or wire length. Other approaches use exact optimization algorithms or metaheuristics.

2.4.1 Deterministic approaches

Cone partitioning

Cone partitioning is a method dedicated to combinatorial circuits, defined for the first time by G. Saucier *et al.* [172]. In cone partitioning, a combinatorial circuit is represented as a graph and it can be extended to that hypergraph or red-black hypergraph.

A *cone* is the subset of all vertices from which a specified sink vertex is reachable. A cone is a structure emanating from combinatorial circuits representing a connected subcircuit from an output of the circuit to the accessible inputs. An example is shown in Figure 2.1. One way to identify a cone in a circuit is to compute the set of vertices reachable from a sink vertex. In this way, all source vertices reachable from the sink vertex are in the same cone, as are all vertices between the source vertices and the sink vertex. That is, the number of cones is equal to the number of outputs in the circuit.

This approach makes it possible to embed critical paths that end in the sink vertex associated with the cone. However, cones can share multiple vertices, e.g. in Figure 2.1, both cones share a common source vertex. If two cones share at least one vertex, then they share at least one source. In fact, since a cone is a set of vertices that can be reached from a sink vertex by traversing the vertices back to the source in the opposite direction of the arcs, the only way to get one or more shared vertices for at least two cones is to have a source vertex that reaches at least two sink vertices. In the context of cone partitioning, each cone defines a part. However, when two cones share vertices, we need to define a selection strategy to determine which cone will be cut or not. A cone is cut if some of its vertices are in another part.

In [172], the authors define several criteria for cutting cones. For example, if two cones have a non-empty intersection, the cone with the highest criticality is not cut. The authors extends their work for partitioning circuit for multi-FPGA platform as in [26, 27].

Integer Programming

Linear programming was developed in the 1940s and 1950s by researchers such as G. Dantzig [52] and L. Kantorovich *et al.* [53]. It has become a powerful tool for solving optimization problems where a linear function needs to be maximized or minimized under linear constraints.

However, in many cases, the optimal solutions to linear programming problems are fractional numbers [84]. This was often impractical or unacceptable in domains where variables had to represent discrete or integer values, such as resource allocation, scheduling, and network design. Thus, integer linear programming became

a natural extension of classical linear programming.

Over the following decades, several advances have been made in integer linear programming. In the 1960s, A. H. Land *et al.* presented the branch-and-bound method [120], which divides fractional solutions into smaller subproblems so as to explore possible integer solutions. In the 1970s, Richard M. Karp proved that integer linear programming is an NP-hard problem [105], meaning that there is currently no polynomial algorithm that can optimally solve all cases efficiently. Despite the inherent complexity of integer linear programming, many researchers have continued to develop efficient methods and algorithms to solve it. Techniques such as dynamic programming [105], branch-and-cut methods [144, 98], and genetic algorithms [100, 99], have been successfully developed and applied. A. Henzinger *et al.* [89] presented in their work an algorithm for graph partitioning based on integer linear programs. Since those programs cannot scale well to large graphs, they proposed an adaptation of integer programs to heuristically improve partitions. Recently, I. Bustany *et al.* [31] used an integer linear program as initial partitioning step inside a multilevel scheme. The objective function is to minimize the cut size.

Clustering approaches

Clustering is a form of partitioning where the number of parts is not fixed; only the capacity for each part is limited. However, it is possible to translate from a partitioning formulation to a clustering formulation. The only difference between the two formulations concerns the balancing factor. In the case of partitioning, the number of parts is associated with a maximum capacity per part, assuming that each part is filled as evenly as possible, according to a balancing factor. In clustering, however, only the maximum capacity of the parts is limited, but the number of parts is not. It is therefore possible for a cluster to consist of a single element. As electronic circuits have grown in size, they have become increasingly challenging to manage in practice. As a result, reducing the apparent size of instances has become a major goal for VLSI design. Clustering algorithms are one of the approaches that have been studied to reduce the apparent size of circuits to make them more practical for automatic processing.

E. L. Lawler *et al.* [122] have presented a polynomial algorithm for grouping the vertices of a circuit so that the resulting delay is optimal. The authors use vertex replication to reduce the number of cut paths in the circuit. A trivial way is to replicate the circuit in each part if capacity permits, thus avoiding cutting the critical path. Node replication makes the problem solvable in polynomial time. C. Sze *et al.* [185] presented an extension the clustering algorithm to a more general delay model. Other works have proposed clustering approaches that also take advantage of vertex replication [157, 190].

More recently, Z. Donovan *et al.* [60, 61, 62] have studied the combinatorial

circuit clustering problem with and without vertex replication. They propose several algorithms to solve this problem. The authors present NP-hardness proofs for the DAG circuit clustering problem with minimization of critical path degradation during the clustering step, e.g., minimization of the number of clusters along the most critical paths. They propose exact exponential algorithms and approximation algorithms parameterized by cluster size. Further details of this work can be found in Z. Donovan's thesis [59].

Other work on combinatorial circuit clustering is available in these papers [145, 47]. More details about clustering methods for circuit partitioning can be found in Chapter 4.

Multilevel scheme

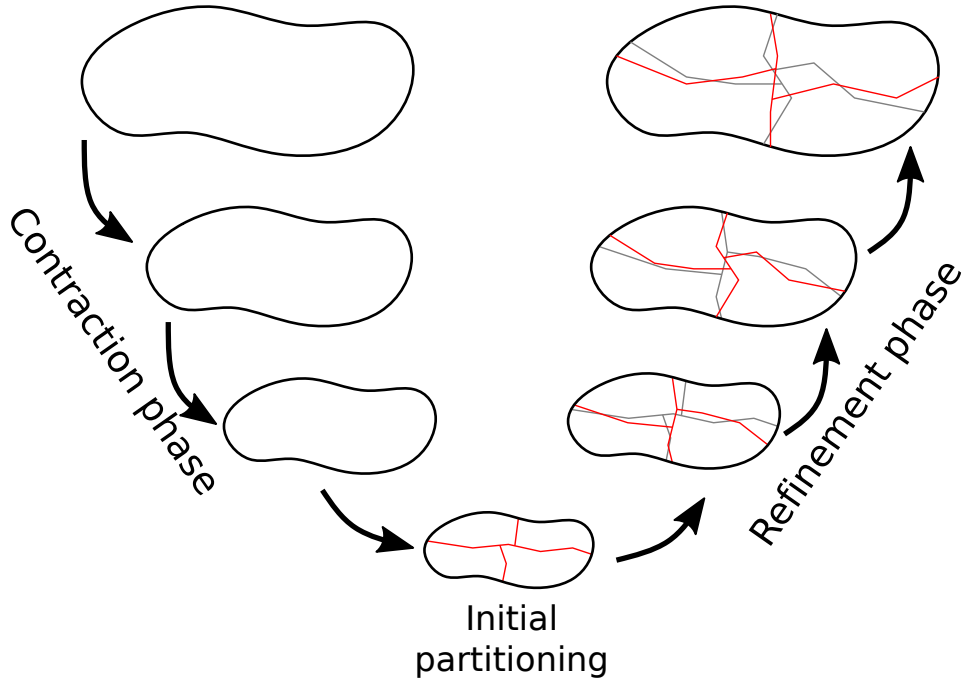


Figure 2.2: The multilevel scheme which consists of three phases: *coarsening*, *initial partitioning*, and *refinement*.

The multilevel scheme has proven to be a very efficient methodology for partitioning graphs and hypergraphs. The multilevel scheme was first introduced by S. T. Barnard and H. D. Simon [19] in 1994 for the graph bisection problem. Independently, T. N. Bui *et al.* [29] used a multilevel heuristic for sparse matrix factorization. B. Hendrickson *et al.* [88] developed a multilevel algorithm for graph partitioning, and S. Hauck *et al.* [87] for partitioning logic circuits for VLSI de-

sign. J. Cong *et al.* [46] have also proposed a multilevel method for VLSI design in which circuits are represented by graphs and a clique-oriented clustering algorithm is designed for the thickening stage. The clique-oriented clustering algorithm favors the clustering of vertices that form cliques or semi-cliques in the graph. An illustration of the multilevel scheme can be found in Figure 2.2.

The multilevel framework consists of three phases: *coarsening*, *initial partitioning* and *refinement*. The coarsening phase recursively uses a clustering method to transform the considered hypergraph into a smaller one. The aim of good clustering algorithms is to try to preserve the same global structure at each clustering level, but it is hard to achieve it in practice. During the second phase, an initial partitioning is computed on the smallest, or coarsest, hypergraph. Then, in the third phase, an algorithm is applied at each recursion level to prolong the computed partition to the upper level, and subsequently refine it. Let us recall that the most common algorithms used for the refinement phase are the Kernighan-Lin (KL) [114] and Fiduccia–Mattheyses (FM) [75] algorithms. As described in the previous Chapter, these two heuristics are based on *local search* to move vertices across parts, so as to reduce the cut of balanced hypergraph bipartitions. KL selects a pair of vertices from each part of a given bipartition which maximize swap gain. Here, swap gain refers to the reduction in the number of cut edges. FM computes a move gain for each vertex, and performs a single move at each iteration. More details about algorithms integrated in multilevel schemes can be consulted in a survey on hypergraph partitioning which has been recently produced [35].

The netlists produced by synthesis tools can be organized hierarchically, with the overall circuit being organized as a set of functional blocks, which in turn are recursively made up of sub-blocks. The lowest blocks in the hierarchy consist only of basic blocks. Another strategy is to use the circuit hierarchy to define the different levels. Hierarchical partitioning starts with the specification of the overall circuit, which is divided into smaller sub-circuits that can be individually designed and optimized. The efficiency of the decomposition methods depends on the applications. H. Krupnova *et al.* [118] presented in their work a hierarchy-driven circuit partitioning for large ASICs prototyping. D. Behrens *et al.* [20] used the circuit hierarchy as the clustering criterion. As the circuit hierarchy is part of the input data, the computation time of the partitioning flow is reduced compared to k-way partitioning tools. Y. Cheon *et al.* [39] introduced a multilevel partitioning algorithm guided by the circuit hierarchy and obtains better results than hMetis. More recently, U. Farooq *et al.* [73] presented hierarchy-based circuit partitioning for a multi-FPGA platform. Their experiments compare mono and multi-cluster CPU circuits generated with the open source tool DSX [154]. Mono-cluster benchmarks are mainly characterized by non-hierarchical interconnection. Multi-cluster benchmarks are hierarchical in nature, with different clusters inter-

connected in a hierarchy. The results show that the multilevel approach produces better results than the hierarchical approach for the mono-cluster benchmark than for the multi-cluster benchmark. On the opposite, in their results, the hierarchical method performs better for the multi-cluster benchmark.

Other works also use hierarchical circuit properties to partition circuits such as those of W. J. Fang *et al.* [72] and, more recently, U. Farooq *et al.* [74].

min-cut based approach

The partitioning of graphs and hypergraphs has been a well-studied topic in recent years and remains so today. Several tools have been developed, each with its specific focus, such as computational speed [106, 109, 34], quality of results [141, 169, 173], or the specificity of the problem [149]. For most hypergraph partitioning tools, the cost function to minimize is the cut size. However, C. Ababei *et al.* [4] have shown that this function does not model properly the side effects associated with critical path degradation. In the previous chapter, we evidenced that partitioning, which does not consider the target topology, cannot model either the critical path degradation associated with routing costs.

This is why, several works have coupled pre- and post-processing algorithms with min-cut tools to take into account critical path degradation during the partitioning phase.

J. Cong *et al.* [45] introduced an algorithm called Hierarchical Performance driven Multilevel Partitioning (HPM), which addresses two objectives simultaneously: cut size and delay minimization. To achieve delay minimization, the authors also use retiming methods. Retiming is a method that allows the petitioner to modify the circuit structure. The results of the HPM algorithm provides better delays than those of hMetis and FLARE [45]. FLARE is a performance-oriented circuit k-partitioning algorithm with acyclicity constraints. The acyclicity constraints are interesting, e.g., to avoid a path that traverses more than k parts. Readers interested in acyclic partitioning can refer to [44, 140, 91, 92, 153]. An approach similar to bipartitioning is proposed in a paper by A. B. Kahng *et al.* [104], in which they define the notion of a V-shaped vertex. Given a vertex v along a path, if its predecessors and successors are in the same part, which differs from that of v, then v is V-shaped. The goal is to minimize this type of situation, which increases the path's criticality because of cut costs.

S. Ou *et al.* [143] presented an algorithm for circuit bipartitioning with timing constraints and node replication, using iterative quadratic programming (TPIQ). TPIQ_P, an extension of TPIQ which include placement constraints, is proposed in [142].

C. Ababei *et al.* [4] devised an algorithm, based on hMetis, to minimize cut size and critical path degradation. The authors use Elmore's model to extract

the timing associated with the circuit. The Elmore delay for an arc a, from a net source to one of its sinks, is defined as:

$$\text{Delay}(a) = R_e \times \left(\frac{C_e}{2} + C_t \right) \quad , \tag{2.1}$$

with R_e is the wire lumped resistance, C_e is the wire lumped capacitance, and C_t is the total lumped capacitance of the source node of each net. From this timing, they evaluate a subset of critical paths that are the most critical of the circuit. The length of the critical paths is used as an hyperedge weight, to prevent these hyperedges from being cut. The set of most critical paths is regularly re-evaluated because it can be modified when adding cutting penalties.

S. H. Liou *et al.* [127], presented a partitioning flow with post-processing and pre-processing around the hMetis algorithm to account for the performance degradation associated with the partitioning step. The pre-processing consists in placing the circuit on a 2D surface. The distance between vertices is used as a weight for each hyperedge during the partitioning stage with hMetis. The aim is to avoid cutting a pre-evaluated set of paths that are considered critical. Finally, a post-processing step is applied to optimize the assignment of the parts on the multi-FPGA platform. The authors also perform an optimization pass for signal multiplexing. U. Farooq *et al.* [73] compared two circuit partitioning methods for multi-FPGA platform prototyping; one is based on a multilevel approach, and the other on a circuit hierarchy.

M. H. Chen *et al.* [38] proposed a partitioning algorithm, similar to hMetis, which considers a metric associated with the distance of the vertex distance in the hypergraph. This metric can be approximated by the eccentricity, *i.e.*, the greatest distance between a node and other nodes in the hypergraph. Eccentricity is used for the coarsening step. The authors proposed a routing indicator, by calculating a route between vertices, using an A^* algorithm. From these indicators, the authors propose a partitioning flow, trying to minimize the impact of partitioning on the circuit's performance.

D. Zheng *et al.* [193] presented TopoPart, a topology driven hypergraph partitioner designed for targeting multi-FPGA platform. TopoPart first applies an algorithm which find candidate FPGAs for each vertex while respecting topology and fixed vertex constraints. The coarsening step is then performed by merging different vertices that not only have high connectivity but also share a higher number of FPGA candidates. Next, an initial partitioning is performed on the coarsest hypergraph to obtain a feasible initial solution. The initial partitioning assigns first fixed vertices, if exists. Then it assigns neighbors of fixed vertices and vertices with the lowest number of FPGA candidates. Finally, uncoarsening and refinement steps are applied to minimize the cut size while maintaining topology and resource constraints.

TopoPart is designed to take topology into account when partitioning a circuit for a multi-FPGA platform. TopoPart partitions a circuit while minimizing the distance between vertices within the topology and the size of the cut.

I. Bustany *et al.* [31] introduced TritonPart, a recent hypergraph partitioner. TritonPart also used a multilevel scheme to tackle large hypergraphs. In the coarsening step, TritonPart uses the First Choice algorithm [112] with a merge weighting function which combines the heavy edge evaluation defined in [173] and the sum of the hyperedge timing costs divided by their size. Timing cost is extracted from a set of finite paths of the circuit. After coarsening, TritonPart applies an integer linear program to the coarsest hypergraph which optimize the cut size. A starting solution is computed before by applying a random partitioning as in [107] followed by VILE algorithm [32]. An additional delay is added when two adjacent vertices in a path are not in the same part. This delay is constant and does not take into account the part connectivity of the target topology. Finally, TritonPart performs uncoarsening and refinement steps. The refinement applies a local search algorithm based on FM [75], which optimizes a cost function which is a combination of cut cost and path cost. Path cost is updated during each refinement process. The authors also introduced a greedy refinement algorithm that randomly visits all hyperedges and tries to move a hyperedge's subset of vertices, from one part to another, without violating the balance constraint.

TritonPart tackles timing cost degradation during partitioning by incorporating timing cost, in addition to cut cost, in their objective function.

Other works that deal with the partitioning problem, taking into account the criticality of the paths, are [58, 158, 163, 66, 67, 134, 43, 2, 3, 182, 184, 188].

2.4.2 Probabilistic approaches

Probabilistic approaches such as simulated annealing and tabu search have also been studied for graph and hypergraph partitioning. Some of these approaches are especially concerned with critical paths. The function to be minimized is not exactly the function f_p defined in the previous chapter, since the target topology is not always taken into account and the model is a classical hypergraph. This subsection describes and details state-of-the-art probabilistic approaches to circuit partitioning.

Simulated annealing

Simulated annealing is a local search-based metaheuristic introduced in 1983 by S. Kirkpatrick *et al.* [115] in 1983. The physical process of annealing in metallurgy inspires this algorithm.

Some parameters are required to simulate the physical process and address the problem to be optimized. These parameters are listed below:

- T the temperature
- S the solution
- E energy of the system
- Δ_E energy variation

Simulated Annealing algorithm simulates the annealing process at a temperature of T. The process starts from an initial state S_0 at a corresponding temperature T_0. For each state correspond a energy cost E. During the process, the temperature is slowly reduced in a geometric progression or step by step. The process consists of successive *epochs* in which the temperature is fixed and a fixed, large enough number of solutions are evaluated. At the end of an epoch a new, lower temperature is determined, following an exponential reduction factor or by a fixed decrease value. Hence, at each temperature level T_i, a new state S_i is explored.

When a new candidate solution is produced, the energy variation Δ_E, compared to the current solution is evaluated. The acceptance of the candidate as the new current solution is decided by the condition:

$$r < e^{-\Delta_E/T_i} \quad . \tag{2.2}$$

With $r \in [0, 1]$ a pseudo-random number uniformly chosen. If the condition is valid, S_i becomes the new current solution. Note that accepting a state of higher energy allows the algorithm to explore a more significant part of the space states. However, if S_i has a lower energy than the precedent current solution $S_{i'}$, then $\Delta_E < 0$, thus, $e^{-\Delta_E/T} > 1$, and S_i is the new current solution. In this case, the new solution improves the criterion and the algorithm moves towards the optimum in the neighborhood of the precedent state.

Note that at the beginning of the process, the temperature T is high, that is, the value of $e^{-\Delta_E/T}$ tends to 1. This favors diversification in the exploration of the states space. Then, as the annealing process progresses, the algorithm tends to intensify the search within the most promising areas of the search space.

K. Roy-Neogi *et al.* [163] presented an optimization algorithm for circuit partitioning based on the principle of simulated annealing. In the context of their work, circuits are partitioned for prototyping on multi-FPGAs platforms. The authors adopt a dynamic weighting process to estimate the effect of partitioning on the circuit delays. Other works based on simulated annealing approaches have addressed

the problem of partitioning and circuit placement [132, 79, 131]. Simulated annealing is often used for the placement and routing when the circuit is partitioned. For example, in their paper, P. Maidee *et al.* [130] presented a partitioning-based algorithm for placement on multi-FPGAs platform. This algorithm first partitions the circuit using hMetis [107], and then uses the VPR [23] simulated annealing-based tool to refine the placement.

Tabu Search

Formalized in 1986 by F. Glover [81], tabu search is a local search algorithm that maintains a FIFO queue, called *tabu list*, of solutions Q_s that have already been explored. Starting from a solution S, the algorithm explores the neighborhood of S, while excluding the solutions in the queue Q_s. This method avoids returning too quickly to a solution that has already been explored. For minimization algorithms, if there is no better local minimum cost solutions in the neighborhood of S, the search continues by exploring higher cost solutions. This makes it possible to escape from a local minimum.

The tabu list size must be adapted to the objective, the nature of the problem, and the expected performance. The size of the queue has significant impact on the computation time of the algorithm. However, it is possible to re-explore solutions in the tabu list using the "aspiration" value. This value is an acceptance metric based on the cost or properties of the solution. Aspiration can be used, for example, to encourage the exploration of desired type of solutions.

S. Areibi *et al.* [13, 14] applied the tabu search strategy to a simulated annealing algorithm in their papers. J. M. Emmert *et al.* [69] presented a two steps approach, partitioning and placement, both using tabu search algorithm. For the partitioning step, the algorithm minimizes the cut size. Then, for the following placement step, the algorithm minimizes the total Manhattan distance by placing the circuit on a grid. Other works on tabu search addressed the problem of partitioning circuits with or without considering the target topology [128, 68, 166].

Ant colonies

M. Dorigo *et al.* first introduced an ant colony optimization (ACO) algorithm [63, 42] in 1991. An ant colony algorithm is inspired from real ants' behavior as they forage for food sources and communicate with each other by depositing pheromones.

The first step consists in building the solutions: the ants move along a graph representing the problem to be solved. The ants build solutions sequentially, following specific rules, moving from one node to another.

An important aspect in ACO is the pheromone deposition strategy. Once

a solution has been constructed, the ants deposit pheromones on the borrowed edges. Pheromone levels are adjusted according to the quality of the solution, usually measured by a dedicated objective function and constraints.

Pheromone levels on edges are updated according to specific rules. For example, edges taken by ants that have built good solutions can increase their pheromone levels, while edges taken by ants that have built bad solutions can decrease their pheromone levels. The steps of building solutions, depositing pheromones, and updating are repeated over several iterations. As the iterations progress, the pheromone levels guide the ants to favor the best-quality solutions. The algorithm gradually converges to one or more local minimum solutions.

ACO is a heuristic method for combinatorial optimization problems, such as the Traveling Salesman Problem (TSP) studied in the work of M. Dorigo *et al.* [64]. An ACO approach for graph bipartitioning can be found in a work done by M. Leng *et al.* [124]. A k-way graph partitioning approach using ACO is studied in work by K. Tashkova Korošec *et al.* [187]. There exist studies for netlist and hypergraph partitioning. For instance, the work by P. Danassis *et al.* [51] introduced a novel netlist partitioning and placement algorithm named ANT3D, targeting 3-D reconfigurable architectures based on ACO. More recently, R. Guru Pavithra *et al.* [147] presented an ACO-based partition model for VLSI physical design.

Applying an ant colony algorithm to partition digital electronic circuits for VLSI design permits problem-specific adaptations and custom parameters to account for domain constraints and objectives compared to min-cut tools. Complementary techniques can also be used with the ant colony algorithm to improve partitioning performance and results [51, 147].

Evolutionary Algorithms

One subset of evolutionary algorithms are genetic algorithms (GA) [100, 99] that were originally developed in the 1960s by John Hollande and his colleagues at the University of Michigan. The principle of the algorithm is an iterative process that maintains a population of solutions. Each solution is represented by a string of digits, or *chromosome*. Each string is made up of characters and genes which correspond to the digits in the string. From these digits, each solution has a corresponding cost according to a cost function to optimize.

The heart of the algorithm is to produce multiple generations of populations, *i.e.*, sets of solutions. Each generation corresponds to one iteration of the algorithm. During each generation, the solutions in the current population are evaluated by a cost function dedicated to the optimization problem. Based on these evaluations, a new population of candidate solutions is formed using specific genetic operators: crossover and mutation. Crossover consist of combining the genetic information (binary string) of two parent strings to generate new offspring.

The mutation flip an arbitrary bit of *chromosome* from its original state. Mutation introduce diversity into the sampled population and it is used in an attempt to avoid local minima.

S. M. Sait *et al.* [165] addressed the problem of optimizing delay, power and cutset in the partitioning step at the physical level. In their works, the authors presented three iterative approaches based respectively on a genetic algorithm, a tabu search and a simulated evolution to solve the multi-objective optimization problem of partitioning. S. S. Gill *et al.* [80] addressed the k-way circuit partitioning using genetic algorithms. J. I. Hidalgo *et al.* [96] proposed a genetic algorithm for partitioning and circuit placement for multi-FPGA platforms. Their work models the circuit as a graph and targets a mesh topology of size 4, made up of 4 FPGAs. The algorithm minimizes the number of inputs and outputs connecting each FPGA, while preserving the circuit structure, *i.e.*, connections and cells.

Another type of evolutionary algorithms are memetic algorithms. Memetic algorithms combine a genetic algorithm with a local search algorithm to improve convergence. For example, S. Areibi's papers [12, 15] presented a genetic algorithm coupled with two local search methods. The first local search method extends the Fiduccia and Mattheyses algorithm [75] to k-way partitioning. The second method is an extension of the Sanchis KFM implementation [167] that apply movement when no further improvement exists. This improvement avoids getting stuck in a local minimum, improving the convergence of the algorithm. Other works on the circuit partitioning problem, based on evolutionary algorithms, can be found in [17, 170, 18, 117, 181].

2.5 Conclusion

This chapter overviewed the current state of the art in circuit partitioning with path length minimization. The complexity section introduced the fact that the partitioning process is NP-hard. This means that effort must be made to develop efficient heuristics to compute a good partition in an acceptable computation time. Some publicly available tools have been developed, which are presented in Section 2.3. However, these tools are dedicated to the problem of cut minimization. This problem is relevant to us, but it is not our main objective. Relevant works are presented in Section 2.4.1. These works are based on min-cut tools for circuit partitioning, that is, the authors presented kind of processing to try to model the path costs to drive the min-cut tool. Moreover, other works presented in Section 2.4.2 are based meta-heuristics algorithms. These algorithms requires to specify good parameters and an extra effort to find a good embedding of the problem. Hence, these processes for using min-cut tools or meta-heuristics, are an extra effort that is not necessary if you use a dedicated algorithm. As Abraham Maslow said:

> 'If the only tool you have is a hammer, it is tempting to treat everything as if it were a nail."

For these reasons, in this thesis we propose a dedicated hypergraph structure to model circuit properties such as register and combinatorial cells, register-register paths, and acyclicity in combinatorial blocks. Dedicated clustering, partitioning, and refinement algorithms based on this new hypergraph structure will also be presented in the following chapters.

Chapter 3

Experimental setup and methodology

This chapter presents the methodology used to evaluate the path cost function f_p of partitions of the red-black hypergraphs.

Measuring f_p means computing the critical path in some red-black hypergraph. The problem of computing the longest path in a hypergraph is generally intractable, due to its NP-hardness. However, because of the properties of red-black hypergraphs, such as the acyclicity within the DAHs that compose them, one can compute the cost function f_p in polynomial time. The algorithms needed to compute f_p are presented in Section 3.1.

In Section 3.2, we will present the digital circuits we used to compare our proposed partitioning strategies with solutions based on min-cut partitioning tools. We will use two sets of publicly available benchmarks and two sets created by us. These third and fourth sets of benchmarks have been designed to contain topologies of circuit with characteristics that differ significantly from those of the first two sets of benchmarks.

In this thesis, we are interested not only in preventing degradation of the critical path during partitioning, but also in the cost associated with routing signals across parts. Therefore, it is necessary to evaluate partitioning strategies on different multi-FPGA platform topologies. To this end, in Section 3.3, we will uses four platforms, two of which consist of four parts, and the other two of eight parts.

3.1 Critical path in red-black hypergraphs

In this section, we present the problem of computing the critical path in a red-black hypergraph. Note that when modeling digital electronic circuits, vertices are associated with a traversal time. As presented in Section 1.2, this time represents the delay of a cell in a digital electronic circuit. By adding up these delays along a path between two registers, *i.e.*, red vertices, we obtain the maximum total traversal time, *i.e.*, the path cost f_p. This computation time is generally determined by the technology available on the target FPGA board. Sometimes, the logical depth is used to evaluate the path cost, as in [127, 38]. In this dissertation, we do not deal with specific industrial topologies. Therefore, we set an equal computation time per cell for all vertices. This weighting brings us closer to the logical depth metric. Logical depth is equivalent to the length of a path, *i.e.*, the number of vertices traversed along the path.

Counting the number of traversed vertices is also known as the longest path problem. In Subsection 3.1.1, we will define the general longest path problem for graphs and hypergraphs, which is NP-hard in the general case. In this thesis, we address the problem of mapping a red-black hypergraph onto a non-uniform topology. Consequently, extra-cost due to routing has to be integrated in the cost function f_p defined in Chapter 1. In Subsection 3.1.2, we will prove that the longest

path problem for red-black hypergraphs can be solved in polynomial time. Hence, we will define an algorithm to compute the cost function f_p of a partition of a red-black hypergraph. This algorithm will be used to evaluate all the partitioning results presented in this dissertation.

3.1.1 Hypergraphs

Computing the critical path of a hypergraph amounts to computing its longest path. A *longest path* between two vertices u and v is a simple path of maximum length between u and v. Formerly, we defined $d_{\max}(u, v)$ as the length of the longest path between u and v. Given a hypergraph $H = (V, E)$, the longest path problem consists in finding a path p in H such that:

$$d(p) = \max_{u,v \in V} \{d_{\max}(u, v)\} \ . \tag{3.1}$$

The longest path problem is known to be NP-hard in the general case [78]. Consequently, f_p cannot be computed in polynomial time, nor can it be solved for any cost model that operates on hypergraphs or graph representations.

However, in the next subsection, we will show that, for red-black hypergraphs, the longest path problem can be solved in polynomial time.

3.1.2 Red-black hypergraphs

In [177], R. Sedgewick and K. Wayne presented an algorithm for computing a critical path in polynomial time on acyclic graphs based on Kahn's algorithm [103]. Since a red-black hypergraph is composed of one or more DAHs, the longest path problem can be solved in polynomial time in a DAH, by applying the algorithm in [177], which topologically sorts the DAH vertices. We propose an adaptation of this algorithm to red-black hypergraphs in order to process each DAH separately. We will use this new algorithm, called Algorithm 1, to compute the value of f_p for any red-black hypergraph.

In this dissertation, we are interested in the set of red-red paths, P^R, of red-black hypergraphs. Algorithm 1 computes path lengths by propagating the traversal times from each vertex to its neighbors. Each vertex is processed by traversing the topological sorting of vertices, to ensure that a vertex propagating its maximum local path length has already been updated by all of its incoming neighbors. At the end of the algorithm, red vertices have the value of the longest path ending in them. This data can be used to analyze the number of critical, quasi-critical, and non-critical paths. We denote a path as *quasi-critical* if its length is close to the length of the critical path. Formally, a quasi-critical path p is defined as

Algorithm 1 Compute the longest path in a partitioned/mapped red-black hypergraph

Require: $H = (V, E)$, $d : V \to \mathbb{N}$, a vertex delay function, sort, an array which stores a topological sort of V, is_red, a boolean array indicating if a vertex is red, $\pi : V \to \mathbb{N}$, a partition of H, $D : (\pi, \pi') \to \mathbb{N}$, a partition delay function.
Ensure: $p_{\max}$ maximal length of path in DAHs of H according to a partition.

$p_{\max} \leftarrow 0$
$Q \leftarrow \{\}$
length_fwd $\leftarrow \{\}$
for $u \in V$ **do**
    length_fwd$[u] \leftarrow d[u]$ ▷ The initial length of a vertex is equal to its traversal time
end for
for $i \in \{0, |V| - 1\}$ **do**
    $u \leftarrow$ sort$[i]$ ▷ Process each vertex following a topological sort
    for $v \in \Gamma^+(u)$ **do** ▷ For each successor of u, spread its delay
        if is_red$[u]$ **then** ▷ If u is red, only spread its base delay $d(u)$
            length_fwd$[v] \leftarrow \max(d[u] + d[v] + D(\pi(u), \pi(v)), \text{length_fwd}[v])$
        else
            length_fwd$[v] \leftarrow$
                $\max(\text{length\_fwd}[u] + d[v] + D(\pi(u), \pi(v)), \text{length\_fwd}[v])$
        end if
        if $p_{\max} <$ length_fwd$[v]$ **then**
            $p_{\max} \leftarrow$ length_fwd$[v]$
        end if
    end for
end for
return $p_{\max}$

follow:

$$d(p) > d(p_{\max}) - D \ , \tag{3.2}$$

with $p_{\max}$, the critical path and D a delay.

A hypergraph with few critical paths or no quasi-critical paths is unlikely to be degraded too much if the few existing critical paths are preserved from being cut.

Algorithm 2 is an extension of Kahn's algorithm (topological sort) for red-black hypergraphs. It provides the vertex sorting needed for the critical path computation implemented in Algorithm 1.

3.2 Benchmarks

The hypergraphs of our benchmarks are taken from the ITC99 benchmark [50] presented in Subsection 3.2.1, the Titan23 benchmark [137] presented in Subsection 3.2.2, and the Chipyard benchmark [9] presented in Subsection 3.2.3. To represent different circuit topologies, we selected a representative subset of instances of each benchmark.

For each instance, we use topology data to define a traversal cost $d(v)$ for each vertex v, corresponding to the traversal time of a logic element. In order to get realistic results, we set the cut cost to be at least one order of magnitude higher than the propagation delay of a combinatorial cell.

3.2.1 ITC99

The ITC99 digital circuits are designed to evaluate the effectiveness of circuit testing methods such as Automatic Test Pattern Generation (ATPG) and Design for Testing (DFT). ATPG is a method of automating electronic design for finding an input sequence that distinguishes a correct circuit from a faulty one. DFT consists of integrated circuit design techniques that add testability features to the design of a digital circuit product. The added features facilitate manufacturing testing of the designed digital circuit. More details about this benchmark are available in [50].

Algorithm 2 Topological sort of a red-black hypergraph

Require: $H = (V, E)$, is_red, a boolean array indicating if a vertex is red.
Ensure: sort, an array of topological sort of vertices in V

queue $\leftarrow \{\}$
is_visited $\leftarrow \{\}$
sort $\leftarrow \{\}$
in_deg $\leftarrow \{\}$
for $u \in V$ **do**
    in_deg$[u] \leftarrow \delta^-(u)$
    is_visited $\leftarrow$ False
    if is_red$[u] \wedge \delta^+(u) \geq 0$ **then** ▷ If a red vertex u has out-neighbors, then u is a source of at least one DAH
        queue $\leftarrow$ queue $+ [u]$
        is_visited$[u] \leftarrow$ True
    end if
end for
while |queue| > 0 **do** ▷ While a vertex is not been processed
    $u \leftarrow$ queue.pop()
    is_visited$[v] \leftarrow$ True
    sort $\leftarrow$ sort $+ [u]$ ▷ As u is placed in sort, each of its in neighbors is processed, or u is a source of at least one DAH
    for $v \in \Gamma^+(u)$ **do** ▷ As u is placed in sort, each in-degree of its out-neighbors is updated
        in_deg$[v] \leftarrow$ in_deg$[v] - 1$
        if in_deg$[v] == 0$ **then** ▷ If all of in-neighbors of v are processed, v is inserted into the queue
            queue $\leftarrow$ queue $+ [v]$
            is_visited$[v] \leftarrow$ True
        end if
    end for
end while
return sort

Table 3.1: Characteristics of the ITC99 benchmark instances, with $|V|$ the number of vertices, $|V^R|$ the number of red vertices, $|A|$ the number of hyperarcs, Δ the maximum degree of vertices, $\overline{\delta}$ the average degree of vertices $\pm$ its standard deviation, Λ the maximum connectivity of hyperedges, and $\overline{\lambda}$ the average connectivity of hyperedges $\pm$ its standard deviation. As we can see, the proportion of red vertices varies from a circuit to another, see, e.g., B02 ($30/8 = 3.75$) and B19 ($233685/9105 = 25.66$).

Instance	$\lvert V\rvert$	$\lvert V^R\rvert$	$\lvert A\rvert$	Δ	$\overline{\delta} \pm$ std. dev.	Λ	$\overline{\lambda} \pm$ std. dev.
B01	51	11	47	8	3.41 ± 1.48	7	2.63 ± 1.52
B02	30	8	27	6	3.27 ± 1.44	7	2.53 ± 1.67
B03	162	40	156	15	3.56 ± 2.08	15	2.74 ± 2.16
B04	751	99	729	41	3.57 ± 2.53	41	2.76 ± 2.59
B05	1031	104	962	26	3.72 ± 2.41	25	2.79 ± 2.32
B06	58	19	50	10	3.38 ± 2.0	10	2.55 ± 2.09
B07	444	61	433	12	3.63 ± 1.72	12	2.79 ± 1.72
B08	185	36	179	13	3.58 ± 1.8	12	2.76 ± 1.68
B09	172	32	169	10	3.56 ± 1.66	10	2.76 ± 1.73
B10	208	36	200	13	3.62 ± 1.85	13	2.77 ± 1.88
B11	783	57	764	25	3.61 ± 2.23	25	2.78 ± 2.26
B12	1078	134	1070	36	3.88 ± 2.49	36	2.94 ± 2.49
B13	367	78	352	12	3.38 ± 1.54	11	2.65 ± 1.61
B14	10124	357	10044	83	3.8 ± 4.71	83	2.89 ± 4.65
B17	32608	1831	32229	150	3.88 ± 3.89	150	2.93 ± 3.92
B18	115803	4584	114575	87	3.78 ± 4.1	86	2.88 ± 4.1
B19	233685	9105	231243	90	3.78 ± 4.07	89	2.88 ± 4.07
B20	20285	603	20204	99	3.82 ± 4.69	98	2.91 ± 4.63
B21	20631	604	20549	99	3.83 ± 4.73	98	2.91 ± 4.68
B22	30027	865	29929	131	3.83 ± 4.63	130	2.91 ± 4.58

Table 3.1 presents some characteristics of our subset of ITC99 benchmarks. This subset consists of representative circuits with small sizes (B01 to B13), medium sizes (B14, B17, and B20 to B22), and very large sizes (B18 and B19).

In this set of benchmarks, we focus on circuits of different sizes and different characteristics to evaluate the impact of algorithms on these circuits. For example, when partitioning small circuits, paths are more likely to be cut multiple times than when partitioning large circuits, because circuits are sparse. This can be inferred from the ratio between the number of vertices in the longest path and the total number of vertices, see, e.g., B01(8/51) and B19(170/233685). However, in practice, small circuits are not placed on multiple FPGAs. These circuits can therefore be considered critical cases for analyzing the behavior of partitioning methods in extreme cases.

Table 3.2 shows the characteristics of the paths of the red-black hypergraph circuits in this benchmark set. Since the number of paths in a circuit can be exponential, we do not evaluate all paths. Instead, we propagate the delay along a topological order using Algorithm 2 described in the previous section. Then, we examine the maximum propagated delay, to calculate the length of the critical path, $p_{\max}$, and to approximate the number of critical paths. Path criticalities give us information about the possible degradation of the hypergraph critical path. Typically, a hypergraph exhibiting homogeneous path criticality indicates that there are no paths way less critical than others. On the contrary, a hypergraph with heterogeneous path criticalities tells us that there are paths that are much less critical than others. Hence, there is a significant probability that some of these paths may be cut without their cost exceeding that of an uncut critical path. A second metric is the path length. As paths are composed of vertices, if the number of vertices along paths is small, it is easy to place them entirely in one part. On the opposite, it is more difficult to place paths made of many vertices in one part. To evaluate the length of paths, we use the same methodology as for the critical path calculation, and propagate the vertex depth as the vertex delay.

Some circuits have fewer critical paths than their number of vertices. For example, circuits B17, B21, and B22 have $\approx \#p_{\max} = 1$. In other words, there is only one red vertex bearing the critical path value, at the end of Algorithm 1. In these hypergraphs, all black vertices have an equal constant delay, that is, only the length of paths is relevant for analysis. The difference between the length of the longest path and the average path length provides an indication of the existence of paths that can be cut without impacting the critical path of the circuit. Note that this argument is also valid when the cut penalty is less than the difference between the cost of the critical path and that of a given non-critical path. For example, circuit B19 has a critical path cost, *i.e.*, maximum vertex criticality, equal to: $97.82 = 0.58 \times 168 + 0.19 \times 2$ and a median vertex criticality, equal to:

Table 3.2: Paths statistics for the ITC99 benchmark, with $p_{\max}$ the critical path, $\approx \#p_{\max}$ a lower bound on the number of critical paths, $\widetilde{d(p)}$ the median maximum delay of paths traversing vertices, $\max \text{length}(p)$ the longest red-red path, *i.e.*, the maximum number of vertices within red-red paths, and $\approx \text{length}(p)$ the approximated average length of red-red paths $\pm$ its standard deviation.

Instance	$p_{\max}$	$\approx \#p_{\max}$	$\widetilde{d(p)}$	$\max \text{length}(p)$	$\approx \text{length}(p) \pm$ std. dev.
B01	3.86	1	3.29	8	3.78 ± 1.9
B02	3.28	2	2.71	7	3.33 ± 1.89
B03	6.18	12	2.71	12	6.07 ± 3.73
B04	16.62	1	9.09	30	9.48 ± 6.91
B05	31.7	2	16.05	56	17.84 ± 14.29
B06	3.28	3	2.71	7	3.57 ± 1.86
B07	18.36	3	8.51	33	12.17 ± 8.01
B08	9.66	4	5.61	18	6.86 ± 4.38
B09	5.6	8	3.87	11	5.77 ± 2.89
B10	7.34	3	5.03	14	5.9 ± 3.49
B11	20.1	1	8.51	36	9.46 ± 7.17
B12	11.4	2	4.45	21	7.5 ± 3.93
B13	11.98	1	3.29	22	6.09 ± 4.32
B14	35.18	2	23.59	62	25.94 ± 10.96
B17	53.74	1	23.59	94	27.72 ± 20.63
B18	95.5	16	22.43	166	28.11 ± 24.91
B19	97.82	32	22.43	170	29.11 ± 26.04
B20	39.24	2	26.49	69	29.88 ± 14.68
B21	39.82	1	27.07	70	29.44 ± 14.9
B22	39.82	1	25.91	70	28.83 ± 14.47

$22.42 = 0.58 \times 38 + 0.19 \times 2$. Therefore, if the cut penalty is set to 10 nanoseconds, we can cut 7 times paths traversing half of the vertices without exceeding its critical path of length 97.82. On the opposite, B01, B02, B06, B09, and B10 have their median path cost close to the critical path length, that is, in these circuits, a majority of paths are critical.

3.2.2 Titan

The Titan benchmark consists of 23 digital circuits taken from a variety of real-world applications. They reflect modern designs of large-scale systems and use heterogeneous resources. The Titan benchmark was created to compare two CAD tools, VPR [23] and Quartus II from Altera (now Intel). More details about the Titan benchmarks can be found in [137].

In our research team, at CEA LIST, the tool used for logic synthesis is Vivado from Xilinx (now AMD). Logic synthesis is a process that translates the behavioral specification of a circuit, written in a hardware description language (typically, Verilog or VHDL) into a netlist that can be implemented on an FPGA or an ASIC. The netlist instantiates logic elements that are available on the desired target technology; in this case, our FPGA model. It is therefore technology-dependent. A netlist synthesized for an Altera platform cannot be implemented on a Xilinx FPGA. Furthermore, there are several non-compatible generations of technologies available from the same vendor. As a result, in this dissertation, we chose to use the Virtex-7 FPGA technology from Xilinx to transform abstract circuits of the Titan benchmark into an synthesized circuit. To define the vertex delay, we use an approach similar to that of [127]. In this paper, S. Liou *et al.* evaluated the topological depth of paths, *i.e.*, they assigned a unit delay to each vertex. In this dissertation, we assign a delay of 0.58 to each black vertex, which corresponds to LUT[1] traversal time for the Xilinx Virtex-7 speed grade 3. As said previously, delays can be inferred from the targeted technology. However, in this dissertation, we restricted our benchmark to six target topologies described in Section 3.3.

Table 3.3: Applications of the Titan benchmark instances.

Instance	Application
`bitonic_mesh`	Sorting
`cholesky_bdti`	Matrix Decomposition
`dart`	On Chip Network Simulator
`cholesky_mc`	Matrix Decomposition
`des90`	Multi μP system
`xge_mac`	10GE MAC Core
`denoise`	Image Processing

[1] A Lookup Table (LUT) is a base element on a FPGA that contains a programmable truth table, and is used to implement combinatorial logic. LUTs can be seen as programmable (sets of) logic gates.

Table 3.4: Characteristics of the Titan benchmark instances, with $|V|$ the number of vertices, $|V^R|$ the number of red vertices, $|A|$ the number of hyperarcs, Δ the maximum degree of vertices, $\overline{\delta}$ the average degree of vertices $\pm$ its standard deviation, Λ the maximum connectivity of hyperedges, and $\overline{\lambda}$ the average connectivity of hyperedges $\pm$ its standard deviation.

Instance	$\lvert V\rvert$	$\lvert V^R\rvert$	$\lvert A\rvert$	Δ	$\overline{\delta} \pm$ std. dev.	Λ	$\overline{\lambda} \pm$ std. dev.
`bitonic_mesh`	272745	109086	340050	169	6.55 ± 10.63	170	4.84 ± 7.29
`cholesky_bdti`	373573	191502	457948	4098	4.76 ± 11.68	4098	3.67 ± 10.73
`dart`	169434	69624	180625	13613	6.76 ± 47.07	13614	4.46 ± 46.95
`cholesky_mc`	154808	82081	191261	1537	4.72 ± 8.29	1537	3.67 ± 6.96
`des90`	135341	54373	170478	197	6.53 ± 10.56	192	4.86 ± 6.97
`xge_mac`	4403	1974	4609	295	6.12 ± 10.39	296	4.11 ± 9.27
`denoise`	283104	35631	369477	9974	8.01 ± 29.43	9969	5.67 ± 28.33

Table 3.4 displays some characteristics of a representative subset of Titan benchmarks used to evaluate the algorithms proposed in this thesis. This subset consists of circuits with practical applications and sizes that differ from those selected in our subset of the ITC benchmark. The average degree δ of the vertices is not high. In other words, the circuits are sparse hypergraphs.

Table 3.5: Paths statistics for the Titan benchmark, with $p_{\max}$ the critical path, $\approx \#p_{\max}$ a lower bound on the number of critical paths, $\widetilde{d(p)}$ the median maximum delay of paths traversing vertices, $\max \text{length}(p)$, the longest red-red path, *i.e.*, the maximum number of vertices within red-red paths, and $\approx \text{length}(p)$ the approximated average length of red-red paths $\pm$ its standard deviation.

Instance	$p_{\max}$	$\approx \#p_{\max}$	$\widetilde{d(p)}$	$\max \text{length}(p)$	$\approx \text{length}(p) \pm$ std. dev.
`bitonic_mesh`	13.14	32	2.71	24	6.85 ± 6.44
`cholesky_bdti`	10.82	6	0.39	20	3.72 ± 2.84
`dart`	35.76	8	1.55	63	7.5 ± 6.42
`cholesky_mc`	10.82	18	0.39	20	3.71 ± 2.92
`des90`	13.14	544	2.71	24	6.95 ± 6.67
`xge_mac`	6.76	4	0.97	13	3.61 ± 2.17
`denoise`	2304.14	8	20.69	3974	540.71 ± 1047.49

These instances have a low proportion of their red vertices ending a critical path. The circuit with the highest number of red vertices in which a critical path ends is `des90`. The Titan instances, like the big ITC instances, exhibit a significant discrepancy between the number of vertices in the longest path and the average path size. This indicates that there is a low proportion of critical and quasi-critical paths in these circuits. Note that circuit `denoise` has the highest critical path and path length, among all circuits.

3.2.3 Chipyard and neural networks circuits

Chipyard is an open-source generator that can be used, for example, to generate *RISC-V Rocket* chip SOCs. Chipyard was used to generate four digital circuit benchmarks, listed in the following tables. These benchmarks were chosen to assess the behavior of partitioning strategies on these specific topologies, in order to broaden the scope of our study. To enrich the diversity of this benchmark, we add two circuits, MNIST and MOBILENET1, both of which being neural network inference circuits. These circuits have been generated by the N2D2 deep-learning

tool from CEA LIST [36], which allows to export a VHDL description of a data-flow hardware circuit implementing a chosen neural network description.

Table 3.6: Characteristics of the Chipyard benchmark instances, with $|V|$ the number of vertices, $|V^R|$ the number of red vertices, $|A|$ the number of hyperarcs, Δ the maximum degree of vertices, $\overline{\delta}$ the average degree of vertices $\pm$ its standard deviation, Λ the maximum connectivity of hyperedges, and $\overline{\lambda}$ the average connectivity of hyperedges $\pm$ its standard deviation.

Instance	$\lvert V\rvert$	$\lvert V^R\rvert$	$\lvert A\rvert$	Δ	$\overline{\delta} \pm$ std. dev.	Λ	$\overline{\lambda} \pm$ std. dev.
`EightCore`	337176	97915	368541	554	7.92 ± 12.46	553	5.12 ± 11.99
`mnist`	11872	3612	12953	1695	7.79 ± 41.63	1694	5.01 ± 41.49
`mobilnet1`	365637	168169	426998	10243	7.97 ± 54.42	10243	5.16 ± 54.16
`OneCore`	54880	16114	58790	554	7.96 ± 12.31	553	5.1 ± 11.81
`PuLSAR`	418027	111390	429239	43857	8.45 ± 100.27	43857	5.28 ± 100.22
`WasgaServer`	1622238	403886	1675291	4231	8.13 ± 21.74	4161	5.14 ± 21.39

Table 3.6 presents some of the characteristics of the subset of our Chipyard benchmarks used to evaluate the algorithms proposed in this thesis as well as third-party partitioning tools. This benchmark is composed of several basic circuits whose sizes are comparable to those of the circuits selected from the ITC and Titan benchmarks. Note that `WasgaServer` is the biggest circuit of all benchmarks, with 1622238 vertices, 403886 red vertices, and 1675291 hyperarcs. Its average degree $\overline{\delta}$ is higher than that of the ITC benchmark circuits and similar to that of Titan. Like the digital circuits in the previous benchmark sets, these circuits are sparse.

Table 3.7: Paths statistics for the Chipyard benchmark instances, with $p_{\max}$ the critical path, $\approx \#p_{\max}$ a lower bound on the number of critical paths, $\widetilde{d(p)}$ the median maximum delay of paths traversing vertices, $\max \text{length}(p)$ the longest red-red path, *i.e.*, the maximum number of vertices within red-red paths, and $\approx \text{length}(p)$ the approximated average length of red-red paths $\pm$ its standard deviation.

Instance	$p_{\max}$	$\approx \#p_{\max}$	$\widetilde{d(p)}$	$\max \text{length}(p)$	$\approx \text{length}(p) \pm$ std. dev.
`EightCore`	47.94	48	5.03	84	12.08 ± 12.56
`mnist`	6.18	524	2.71	12	4.09 ± 3.21
`mobilnet1`	16.04	7	0.97	29	4.25 ± 3.76
`OneCore`	47.94	6	5.03	84	12.71 ± 12.46
`PuLSAR`	48.52	118	8.51	85	15.72 ± 10.68
`WasgaServer`	48.52	708	8.51	85	15.62 ± 11.14

The path statistics in Table 3.7 exhibit similar average path lengths and critical path costs for all the selected circuits. However, the median path cost is still quite small, with respect to the cost of the critical path.

3.2.4 File formats for the red-black hypergraph

In this subsection, we introduce a new file format tailored to represent red-black hypergraphs. Currently, hypergraphs are commonly encoded in the HGR and HYGR formats that are used in hMETIS and PATOH, respectively. The difference between these two formats is that the HGR format can only encode one weight per vertex, while the HYGR format can encode multiple weights.

In a red-black hypergraph, a vertex should be identified as red or black. In addition, information about delay and criticality should be stored. The HYGR format can be adapted by using the vertex weight vector to store the color, the

delay and the criticality of vertices. To identify the source of each hyperarc, we consider the first vertex in the hyperarc description to be its source. Hence, we propose to adapt the HYGR format file format to represent red-black hypergraphs.

In the file format used in TopoPart [193], the target topology and the circuit are described. Therefore, testing N circuits onto T target topologies need $N \times T$ files. As a consequence, we prefer to separate the description of the circuit and the target topology in our algorithms to avoid multiplying files.

To compare our algorithms with min-cut partitioning tools, we set a single weight for the vertices, representing resource consumption (e.g., register width), because the HGR file format can record only a single weight per vertex. In order to refrain min-cut partitioning tools from cutting critical hyperarcs, each hyperarc is also weighted with the maximum criticality value of the vertices of the hyperarc (see Section 4.2 about the r^* weighting scheme). These values are computed using the algorithms presented in the previous section.

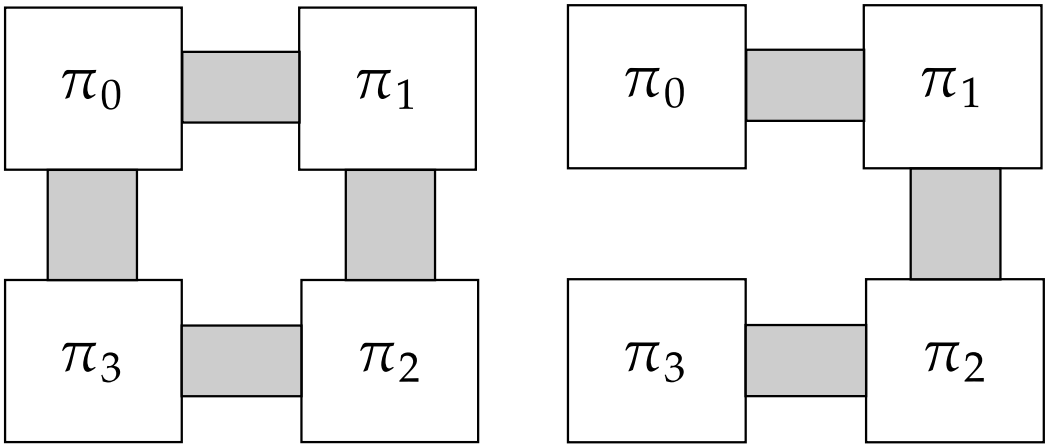
3.3 Target topologies

This section presents the target topologies used in this dissertation for our circuit mapping evaluations. These target topologies were chosen to evaluate the impact of routing cost in circuit prototyping on multi-FPGA platforms. Each circuit is partitioned into the number of FPGAs available on each platform, and placed onto the platform. For example, if a subset of vertices is placed in part 1, then these vertices are considered to be implanted on FPGA 1 on the platform.

Topology-unaware assignments of vertex subsets to parts can result in higher additional routing costs. These additional costs occur when adjacent vertices are mapped onto different FPGAs which are not direct neighbors in the communication network of the platform. In other words, signals exchanged between these two FPGAs must pass through other FPGAs on the platform to reach their destination. To obtain realistic results, we set the cut cost between two connected parts to be at least one order of magnitude higher than the propagation delay of a combinatorial cell.

3.3.1 4-FPGA topologies

In this dissertation, we designed three topologies composed of 4 FPGAs. The first topology, T1, consists of a cycle graph, as shown in Figure 3.1a. The second one, T2, shown in Figure 3.1b, is a path composed of 4 FPGAs. This topology was created to evidence the impact of a partitioner that would not take into account the target topology and the resulting routing costs.



(a) Target topology T1. (b) Target topology T2.

Figure 3.1: Two target topologies T1 (a) and T2 (b) composed of 4 nodes.

Our third target topology, T3, is the complete 4-FPGA graph. It will be used to evaluate critical path degradation, irrespective of additional routing costs due to topology constraints. In this target topology, each FPGA is connected to each other.

3.3.2 8-FPGA topologies

This subsection presents two 8-FPGA topologies, T4 and, T5, that come from the ICCAD 2019 competition [183]. Topology T4 is presented in Figure 3.2. Both topologies have a similar structure. In order to evaluate the impact, on the critical path, of a partitioning strategy that does not take topology into account, we modified the IDs of the parts of T4 to generate T5.

The longest path between two parts of this platform is equal to 7, *i.e.*, the maximum possible routing cost between two vertices is $7 \times D$, where D represents the additional delay for transmitting a signal between two parts.

Like for 4-FPGA topologies, we defined a third topology composed of 8 FPGAs, called T6, in which each FPGA is connected to all the others.

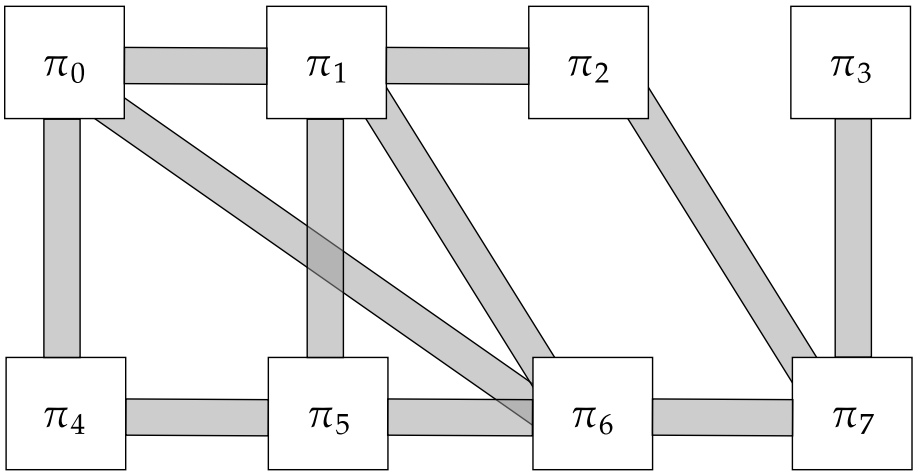


Figure 3.2: T4: the ICCAD 2019 contest target topology for problem B.

3.4 Conclusion

In the first section of this chapter, we presented our method for measuring the quality of a partition for the problem of mapping a red-black hypergraph onto a non-uniform topology. This measure is based on the computation of the longest path within the DAHs that form the hypergraph, including cross-FPGA routing cost. This problem is NP-hard in the general case, but, due to the acyclicity of digital circuit combinatorial blocks, in our case, the computation can be performed in polynomial time.

Since the red-black hypergraph partitioning problem plays an important role in circuit partitioning, we need to evaluate different partitioning strategies on publicly available benchmarks.

As the red-black hypergraph is a new model, defined in Chapter 1 of this thesis, we presented in this chapter an adaptation of the HYGR file format which can encode a multi-valued red-black hypergraph. This file format is used for our experimentations.

To perform our experimentations, we proposed six FPGA platforms, presented in the final section. These platforms consist of three platforms comprising four FPGAs, and three others comprising eight FPGAs.

Chapter 4

Algorithms for coarsening

Coarsening methods are part of the multilevel partitioning scheme. As we have shown in Chapter 2, the multilevel scheme is the most efficient and widely used approach for clustering large hypergraphs. As a result, all modern partitioning tools implement methods based on the multilevel scheme. The general coarsening process consists of a sequence of (hyper)graph shrinking steps, each of which is performed using a clustering method. The aim of a good clustering algorithm is to try to preserve the same global structure at each coarsening level, yet this is hard to achieve in practice.

Clustering also plays an essential role when partitioning electronic circuits, particularly in designing Systems-on-Chip (SoCs) and Integrated Circuits (ICs). Modern electronic circuits are becoming increasingly complex, comprising millions or even billions of transistors. Clustering may be used to reduce the size of a circuit problem by creating blocks of cells. This reduction in size makes circuit problems easier to manage.

This chapter is structured as follows: existing works on clustering for circuits and hypergraphs are presented in Section 4.1. Weighting schemes to measure the attractiveness between nodes for the clustering objective are introduced in Section 4.2. Polynomial algorithms for clustering are presented in Section 4.3; these algorithms are designed for a specific class of hypergraphs. Since the problem is NP-hard, parameterized algorithms are studied in Section 4.4.

Some sections of this chapter are based on our published work [162].

4.1 Clustering

In this section, we present some clustering approaches for graph, hypergraph, and circuit partitioning. The algorithms associated with graph and hypergraph partitioning are generally tailored to optimize cut minimization functions. In the context of circuit partitioning, algorithms should consider additional constraints, as well as an additional objective function aiming at minimizing the degradation of the critical path.

4.1.1 Hypergraph Clustering

Coarsening algorithms are essential for partitioning large hypergraphs. At their core, clustering algorithms reduce the size of the hypergraph partitioning problem by merging vertices according to a matching function. Several functions have been developed to evaluate the quality of vertex merging.

B. Hendrickson and R. Leland [88] propose a *randomized matching* algorithm. This algorithm randomly traverses the vertices, and if the visited vertex is not already matched, a neighbor is randomly selected for merging. The random as-

pect of the algorithm allows it to compute a solution quickly, making it practical for large instances. Another clustering algorithm used for the coarsening phase is called *heavy edge matching*. Like the previous one, this algorithm randomly visits the graph vertices, and if the visited vertex is not already coupled, the algorithm selects a not yet selected neighbor connected by an edge of heaviest weight. G. Karypis and V. Kumar [109] compare randomized matching and heavy edge matching for coarsening algorithms and conclude that the heavy edge matching algorithm provides partitions with better quality and reduces computation time during the refinement stage. Ü. Çatalyürek and Ç. Aykanat [33] proposed an algorithm based on heavy-connectivity matching, that favors merging vertices with highest connectivity. The *connectivity* metric used is also known as the *inner product*. The *inner product* between two vertices is defined as the number of hyperedges shared by these two vertices. T. Heuer and S. Schlag [95] propose a framework for hypergraph coarsening based on the exploitation of community structures in graphs. Their experimental results show that their coarsening method improves the initial partitioning results as well as the final result. Readers can consult the recent survey published by Ü. Çatalyürek *et al.* [35] for more information on clustering methods applied to graphs and hypergraphs.

4.1.2 Circuit clustering when replication is allowed

Combinatorial circuit clustering, when logic replication is allowed, optimises partitioning by duplicating parts of the combinatorial circuit. Let us recall that CA, first defined in Chapter 2.2, is associated with the clustering problem when replication is allowed.

E. L. Lawler, *et al.* [122] have presented a polynomial algorithm for grouping the vertices of a circuit so that the final delay is optimal. The authors use vertex replication to avoid cutting specific paths in the circuit. Other works have proposed clustering approaches that allow vertex replication [157, 190, 185]. A trivial way is to duplicate the circuit in each part if capacity permits, thus avoiding cutting the critical path. Node replication makes the problem polynomially solvable but, if the number of replications is limited, the problem remains NP-hard. A trivial way to prove the hardness of clustering is to study the complexity when replications are limited to 0. However, replication provides a means for reducing the cut cost and delay. An example of replication is shown in Figure 4.1.

In the context of circuit partitioning for multi-FPGA platforms, the capacity is limited by the resource capacity of the FPGAs. If the circuit is small enough, pre-processing can be applied to define a margin based on the capacity of the target platform. Post-processing could then be applied to determine if the circuit has replicable areas, allowing the objective function to be optimized. Critical paths are often so because they contain an important amount of logic, resulting in

significant replication of vertices. As partitioning with replication can be seen as a separate problem, we do not deal with it in this thesis.

4.1.3 Circuit clustering when replication is not allowed

Clustering a combinatorial circuit when logical replication is not allowed aims at optimizing partitioning minimizing f_p under the condition that each cluster is a disjoint subset of vertices. In this thesis, we associate CN with the clustering problem when replication is not allowed. As said, a clustering $\mathcal{C}$ of $H = (V, A)$ is the splitting of V into vertex subsets $\mathcal{C}_i$, called clusters, such that:

(i) all clusters $\mathcal{C}_i$ are given a capacity limit M_i, such that: $\sum_{v \in \mathcal{C}_i} W_v(v) \leq M_i$;

(ii) all clusters are pairwise disjoint: $\forall i \neq j, \mathcal{C}_i \cap \mathcal{C}_j = \emptyset$;

(iii) the union of all clusters is equal to the vertex set V: $\bigcup_i \mathcal{C}_i = V$.

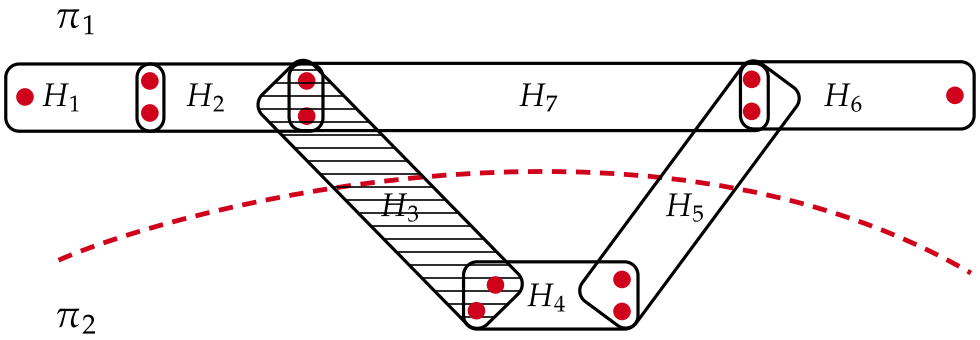
A. A. Diwan *et al.* [57] addressed a similar problem, consisting in placing nodes of a memory access structure on disk pages such that a path through several nodes traverses as few disks as possible. Their data structure is a DAG, and their objective is to cluster the DAG such that the number of shared edges per cluster along a path is minimized. This problem is similar to the unweighted case of the CN problem. The authors also presented a polynomial-time algorithm for trees, and showed that the problem is NP-hard for unweighted DAGs.

More recently, Z. Donovan *et al.* [60, 61, 62] have studied the combinatorial circuit clustering problem, with and without vertex replication. They proposed several algorithms to solve this problem. The authors presented NP-hardness proofs for the DAG circuit clustering problem with minimization of critical path degradation during the clustering step, e.g., minimization of the number of cut penalties along the most critical paths. They proposed exact exponential algorithms and approximation algorithms parameterized by cluster size. Further details of this work can be found in Z. Donovan's thesis [59].

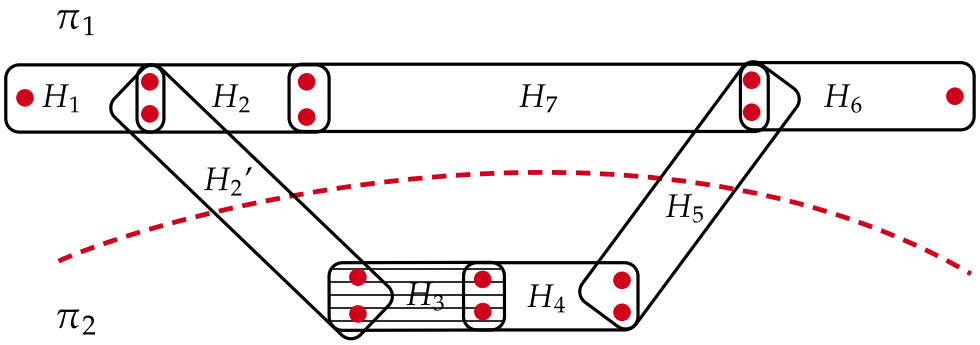
Other work on combinatorial circuit clustering to minimize critical path degradation by placing neighboring vertex pairs in different clusters are available [145, 47].

4.1.4 Conclusion

In this section, we discussed the two clustering problems known as CA when replication is allowed, and CN when replication is not allowed, defined in Chapter 2.2. In the rest of this chapter, we will only consider the CN problem.



a) Partition where DAHs H_3 and H_5 are cut.



b) Partition where H_2' is a replication of H_2 and, DAHs H_2' and H_5 are cut.

Figure 4.1: This figure presents an example of the use of replication to avoid cut a critical DAH. In this example, we have two partitions of a red-black hypergraph composed of seven DAHs in which, the hatched DAH H_3 represents a DAH that contain the critical path. Partition a cuts H_3 and H_5. In partition b, H_2 is replicated in H_2', to avoid the cut of critical path in H_3. Hence, in partition b, H_2' and H_5 are cut and H_3 is fully contained in part π_1.

4.2 Model and weighting schemes

Vertex criticality defined in Chapter 1.2, is used to guide clustering algorithms so as to minimize the impact of partitioning on critical paths [4, 35]. In this section, we will present the various state-of-the-art weighting schemes used to measure vertex criticality. We propose a new weighting scheme that models more finely the criticality which we use to cluster red-black hypergraphs.

4.2.1 Clustering problem model

In the CN problem, an additional constant cost D is added between two neighboring vertices placed in different clusters. However, in our model, the distance between any two vertices u and v (*i.e.*, path cost) may increase during clustering, due to the additional cost that paths have to incur across clusters. Let us recall that the distance function between two vertices in a red-black hypergraph is defined by $d_{\max}(u, v)$, which is equal to the longest path between u and v. Let D be the penalty associated with the distance between two vertices u and v placed in different clusters; the distance function for some clustering $\mathcal{C}$, is thus:

$$d^{\mathcal{C}}_{\max}(u, v) \geq d_{\max}(u, v) + D \ . \tag{4.1}$$

The objective function f_p can therefore be defined as the minimization of the longest path of H subject to clustering $\mathcal{C}$: $f_p = \min d_{\max}(H^{\mathcal{C}})$. We extend the definition of the $CN{<}w, M, \Delta{>}$ problem defined by Z. Donovan *et al.* [59] to red-black hypergraphs as follows:

> *Given a red-black hypergraph $H = (V, A)$, with a vertex-weight function $w : V \to \mathbb{R}^+$, delay function $d : V \to \mathbb{R}^+$, maximum degree Δ, constant D, and a cluster capacity M, the goal is to partition V into clusters such that: (i) the weight of each cluster is bounded by M; and (ii) the maximum delay-length of any red-red path of H is minimized.*

To be consistent with previous definitions of the CN problem, we will keep the Δ parameter, even though we will not use it in the following.

4.2.2 Weighting schemes

As we exposed in Chapter 1.2, the criticality of some vertex v measures the value of the longest path passing through v. Consequently, criticality seems to be an interesting weighting scheme for measuring the attractiveness between two connected vertices. In this subsection, we present three weighting schemes used to guide a clustering algorithm in the context of circuit partitioning with path cost minimization. First, we present the state of the art with respect to weighting schemes, and

show their limitations. Then, we present our weighting scheme based on vertex criticality.

Delay propagation

Several previous works have proposed metrics for clustering, with the objective of path minimization [4, 35]. For example, C. Ababei *et al.* [4] presented a weighting scheme based on delay propagation to drive min-cut tools; the weight between two vertices u and v is equal to the longest path from any red source vertex to vertices u and v. This method calculates local weights along subpaths from red source vertices to any vertex. Thus, within each DAH, $H = (V, A)$:

$$l(u) = \begin{cases} d(u) & \text{if } \Gamma^-(u) = \emptyset \ , \\ d(u) + \max\limits_{v \in \Gamma^-(u)} l(v) & \text{otherwise} \ . \end{cases} \tag{4.2}$$

For any vertex $u \in V$, the value $l(u)$ corresponds to the maximum path cost from any source vertex to u. Therefore, the maximum path cost within some DAH will be found at the level of its sink vertices. A calculation on the subpath does not indicate whether their subpath is on the critical path. Cutting anywhere along a path has the same detrimental effect as adding a penalty to the total path cost. It is to alleviate these issues that the next metric has been devised.

Delay back-propagation

As all critical vertices must be labeled with the same weight, the delay propagation scheme is not adequate. Hence, we have first devised a new weighting scheme based on the back-propagation of path cost:

$$r(u) = \begin{cases} l(u) & \text{if } \Gamma^+(u) = \emptyset \ , \\ \max\limits_{v \in \Gamma^+(u)} r(v) & \text{otherwise} \ . \end{cases} \tag{4.3}$$

For any $u \in V$, the value $r(u)$ represents an upper bound for the path cost of the longest red-red path traversing u. If u belongs to a path of maximum path cost, then $r(u)$ is equal to that path cost.

This weighting scheme accounts better for the overall impact of the cut along a path because, unlike the previous method, the information is back-propagated to all predecessors. However, it may include heavy vertices that do not belong to a longest red-red path, as shown in Figure 4.2. To overcome this problem, we need to define the value of the local critical path through each pair of vertices. For this reason, we have proposed a third weighting system, in the next subsection.

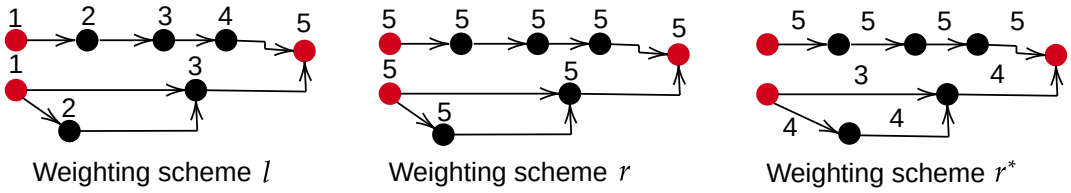


Figure 4.2: An example of the three weighting schemes l [4], r, and r^*. We consider a unit delay for each vertex and a delay equal to zero for each arc. In this example, we can clearly see that scheme l does not effectively weight critical vertices. Scheme r weights critical vertices correctly, but wrongly labels non-critical vertices. Scheme r^* is more relevant in its weighting, with respect to our goal.

Refined delay back-propagation

In this subsection, we present a weighting scheme based on the cost of the local critical path. This scheme retro-propagates critical information throughout the red-black hypergraph and avoids non-critical heavy vertices. The l, r, and r^* metrics are used as weighting schemes, as illustrated in Figure 4.2.

Let $r^*(u, v)$ be the criticality value between connected vertices u and v, defined as follows:

$$r^*(u,v) = \begin{cases} l(u) & \text{if } u = v \ , \\ r(v) - \left(\max\limits_{u' \in \Gamma^-(v)} l(u') - l(u) \right) & \text{otherwise} \ . \end{cases} \tag{4.4}$$

In equation 4.4, $\max_{u' \in \Gamma^-(v)} l(u')$ represents the value of the arcs along the local critical path, which is the longest red-red path traversing v such that, for every other $l(u) < \max_{\forall u' \in \Gamma^-(v)} l(u')$, arcs (u, v) are not in the local critical path. It is a more accurate metric for improving the behavior of clustering algorithms because, in the context of circuit clustering, the aim is to group critical vertices together. If the relationships between vertices correctly reflect criticality, then the clustering algorithm can take advantage of this. An example of the computation of r^* is represented in Figure 4.3.

For each combinatorial sub-circuit modeled with a DAH, the r^* vertex-vertex criticality relation defines a *criticality DAG*. Every hyperarc in the DAH defines a group of arcs in the criticality DAG, in which each arc connects the source vertex to a sink vertex. An example is presented in Figure 4.4. The cut weight of arcs corresponds to the r^* value between source and sink in arcs. Hence, the cut weight of this hyperarc is the maximum of the r^* values between its source and sinks. We will use the criticality DAG in the next section as support for proofs.

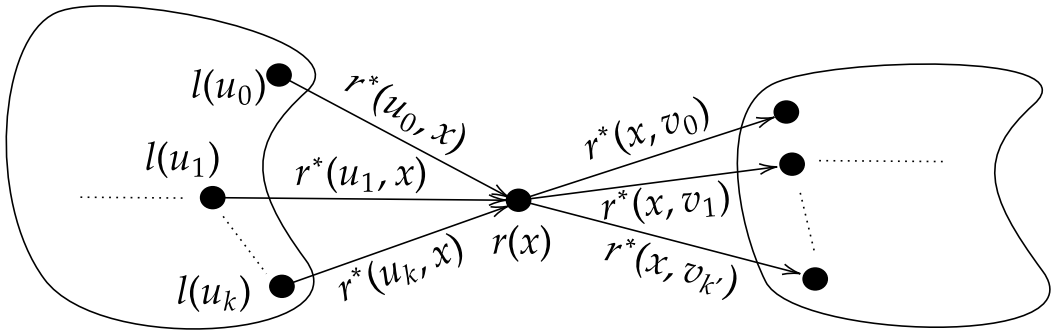


Figure 4.3: This figure exhibits an example of the r^* weighting scheme and how it is computed. $r^*(u_i, x)$ and $r^*(x, v_i)$ are the values of the local critical path between pairs of vertices (u_i, x) and (v_i, x) in this subgraph. There is a maximum value for each $l(u_i)$, $\overline{w}$. For each u_i, $\overline{w} - l(u_i)$ represents the contribution of u_i to the local critical path value $r(x) = \max_{v_i \in \Gamma^+(x)} r^*(x, v_i)$.

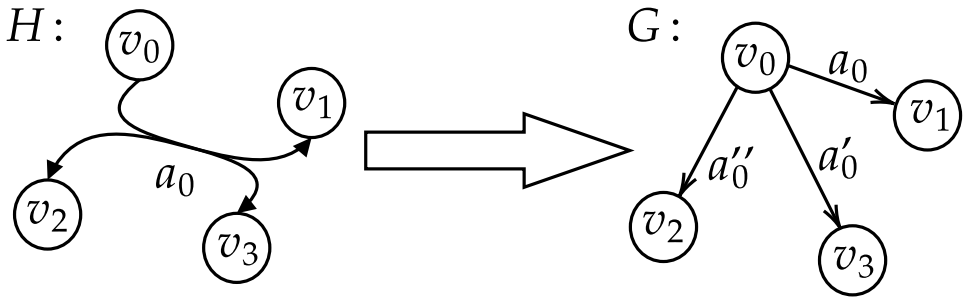


Figure 4.4: This sketch presents an example of vertex-vertex relations within a hyperarc. These relations can be computed in each hyperarc of the hypergraph, and define its corresponding DAG. For any given pair of vertices, multiple arcs resulting from the transformation of multiple hyperedges connecting those two vertices are merged into a single arc whose delay is the maximum of the arc delays.

4.2.3 Conclusion

In this section, three weighting schemes have been defined and compared: l, r, r^*. As a cut anywhere along a path has the same detrimental effect of adding a penalty to the total path cost, critical vertices must have the same criticality. The r scheme back-propagates criticality values to the predecessors, compared to the l scheme, which only propagates delay. Hence, in the l scheme, critical vertices do not have the same criticality compared to the r scheme; that is, the r scheme is better than the l scheme at identifying critical vertices. However, the r scheme can back-propagate the critical value to non-critical vertices, while r^* does not. Indeed, the r^* scheme computes each vertex's local critical path value. In the context of our circuit clustering problem with objective to avoid cuts along critical paths, that is, clustering critical vertices together to avoid possible cuts along critical paths. Consequently, the r^* value appears to be a better model to cluster critical vertices than l and r. The DAH clustering algorithms presented in the following sections use the r^* weighting scheme. We also modeled the critical relation of a red-black hypergraph for the r^* by a DAG. This critical DAG will be used as a support for proofs and explanations to represent the path graph of a DAH with its r^* vertex-vertex criticality relationship.

4.3 Polynomial algorithms for a specific class of DAHs

In this section, we study some classes of hypergraphs for which the CN problem is polynomial. We propose a metric based on the number of intersections between paths, that allows us to identify classes of red-black hypergraphs. We also present an algorithm that solves the problem in polynomial time for critical DAG that does not contain K_3 as a subgraph. Here K_3 refers to the complete graph of 3 vertices. In a complete graph, each pair of vertices is connected.

4.3.1 Path intersection

Z. Donovan *et al.* [61] have shown that, in the general case, the clustering problem CN is NP-hard, and K. Andreev *et al.* [11] have shown that unless $P = NP$, there is no constant approximation factor for balanced graph partitioning in the general case. In this section, we present polynomial algorithms for a specific class of red-black hypergraphs. This class is characterized by the existence of a bound on the number of distinct hyperarcs between two paths sharing at least one vertex. Let

us define two boolean functions f_a and f_b applying to a DAH $H = (V, A)$:

$$f_a(u, v, p, p') = \begin{cases} \text{true} & \text{if } u \in p \cap p', v \in p' \setminus p \ , \\ \text{false} & \text{otherwise} \ , \end{cases} \tag{4.5}$$

with $u, v \in V$ and p, p' a red-red paths in H. Similarly:

$$f_b(u, v, p, p') = \begin{cases} \text{true} & \text{if } u \in p' \setminus p, v \in p \cap p' \ , \\ \text{false} & \text{otherwise} \ . \end{cases} \tag{4.6}$$

If these functions are applied to vertices in hyperarcs $a \in A$, such that $u \in s^-(a)$ and $v \in s^+(a)$, f_a and f_b are valid for each *intersection* u, v between two different paths, p and p'. An *intersection* between two different paths is defined by a couple of vertices u, v, such that, either u is in both paths and v is in p' and not in p (f_a), or v is in both paths and u is in p' and not in p (f_b).

Let p be a red-red path in $H = (V, A)$. Let $\iota(p)$, spelled "iota", be the maximum number of distinct hyperarcs, *i.e.*, counted only once, between a path p and paths sharing at least one vertex with it:

$$\iota(p) = \max_{p' \in P(H)} |\{a \in A : \exists u \in s^-(a), \exists v \in s^+(a), f_a(u, v, p, p') \vee f_b(u, v, p, p')\}| \ . \tag{4.7}$$

An example of the calculation of the above formula is illustrated in Figure 4.5.

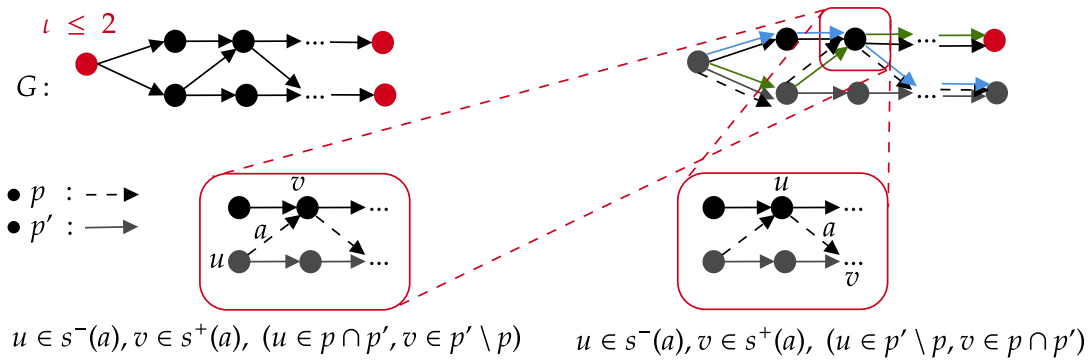


Figure 4.5: In this example, the maximum number of arcs that intersect two different paths is 2. If p is the grey path and p' is the black dashed path, we see that there are two arcs that validate the $(u \in p \cap p', v \in p' \setminus p) \vee (u \in p' \setminus p, v \in p \cap p')$ ι condition, see Section 4.7. Hence, the size of the set is 2, *i.e.*, $\iota(G) = 2$.

Note that, if $\forall p \in P(H)$, $|P(H)| > 1$, and $\iota(p) \leq 1$, then its associated criticality DAG G, in undirected representation, does not have a K_3 minor [159],

i.e., G is a tree. The condition $|P(H)| > 1$ is necessary to overcome the case of the cycle graph C_n which contains one red-red path with a single red vertex. This also holds for directed trees. If $\iota(p) = 0$, the associated weighted DAG is a path graph, a stable graph or cycle graph. Two DAHs H and H' are called $\iota-$*equivalent* iff $\max_{p \in P(H)} \iota(p) = \max_{p' \in P(H')} \iota(p')$. Note that the set of paths in a DAH is the same as in the corresponding criticality DAG. An example of the type of graph structure as a function of ι is shown in Figure 4.6.

4.3.2 A polynomial algorithm for red-black hypergraph clustering

In previous works, M. Goldberg *et al.* [82] and R. M. MacGregor [129] have studied the bisection problem for trees and have proposed algorithms based on dynamic programming. A dynamic programming approach could be interesting in the case where $\iota(p) \leq 1$, $\forall p \in P(H)$. However, since computation time and memory footprint are important aspects in a VLSI context where instances are of large size, we opted for algorithms based on traversal algorithms, such as breadth-first search and depth-first search. A. A. Diwan *et al.* [57] presented a polynomial algorithm to solve the CN problem for trees. Their algorithm traverses the tree from the leaves to the root, creating a cluster for each leaf. If the visited node is an internal node (not a leaf), the algorithm tries to merge this node with its children. A new cluster is created when a processed node cannot be merged with its child cluster without exceeding the maximum cluster size. However, they do not explain how the algorithm optimizes the clusters along the longest path when a vertex cannot be merged with its children. For example, let us consider the case where a vertex v_0 cannot be merged into the cluster containing its children v_1 and v_2 such that v_0 and v_1 belong to the longest path and v_2 does not. In this case, the algorithm creates a cluster containing only v_0, but it would be more optimal here to swap v_2 and v_0 to reduce the number of clusters along the critical path.

The algorithms we propose in this section take advantage of the weighting scheme presented previously. Our r^* weighting scheme allows us to prioritize the clustering of vertices along the critical path, thus minimizing the number of clusters and the number of cuts.

We present below polynomial algorithms based on graph traversal, using the r^* weighting scheme, to solve the CN problem for graphs such that $\iota(p) \leq 1$ $\forall p$.

Theorem 4.3.1. *Let H be a DAH and $G = (V, A)$ be its corresponding criticality DAG, such that $\forall p \in P(H)$, $|P(H)| > 1$, $\iota(p) \leq 1$. There exists a polynomial algorithm to solve the problem $CN{<}[1], M, \Delta{>}$.*

Proof. From Lemma 4.3.2, we show that a DAH and its corresponding criticality

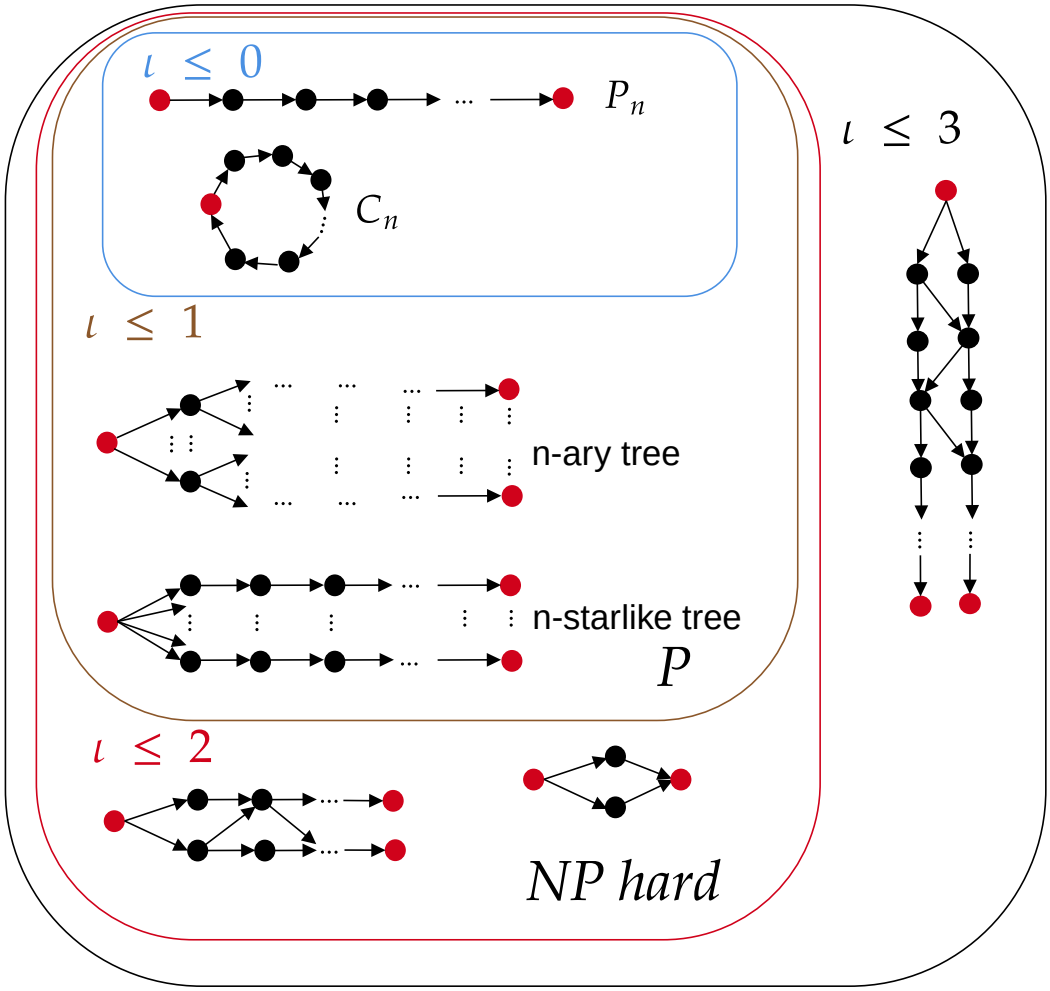


Figure 4.6: This figure shows non-exhaustive examples of graph structures as a function of the ι metric. For any DAG such that $\iota \geq 2$, the CN problem becomes NP-hard.

DAG are ι-equivalent. We show in Lemmas 4.3.3 and 4.3.4 that is it possible to build a polynomial algorithm by calling the procedure associated with Lemma 4.3.3 if $|p_{\max}| \geq M$ (p_{max} cannot be in one cluster); otherwise, the procedure is associated with Lemma 4.3.4. □

Lemma 4.3.2. *Let $H = (V, A)$ be a DAH and $G = (V, A')$ be its corresponding criticality DAG; H and G are ι-equivalent.*

Proof. Let $H = (V, A)$ be a DAH and $G = (V, A')$ its corresponding criticality DAG. Let $p \in P(G)$ be a red-red path defined as a sequence of vertices. For each pair of consecutive vertices (u, v) in a path p, u and v are in a hyperarc $a \in A$, with $u \in s^-(a)$ and $v \in s^+(a)$. By construction of G, $V(G) = V(H)$, and $A' = \bigcup_{a \in A} \{(u, v), u \in s^-(a), v \in s^+(a)\}$. Consequently, all paths defined by a succession of vertices in H are in G and, conversely, all paths in G are in H. □

Lemma 4.3.3. *Let $G = (V, A)$ be a criticality DAG associated with a DAH, $P(G)$ the set of paths of G starting from a source and ending in a sink, such as $\forall p \in P(G)$, $|P(G)| > 1$, $\iota(p) \leq 1$, and $p_{\max}$ be the longest path in $P(G)$. There exists a polynomial algorithm for the problem $CN{<}[1], M, \Delta{>}$ for $M \geq |p_{\max}|$.*

Proof. Since all $\iota(p) \leq 1$, and $|P(G)| > 1$, G does not have a K_3 minor in an undirected representation of G [159]. Let Algorithm A_1 (displayed as Algorithm 3) be the algorithm working as follows: Create clusters by successively traversing the vertices in a predecessor-successor order. The next chosen successor is a vertex which is not already in a cluster, and which is connected to the current vertex with a heaviest arc. Successors that are not chosen are placed in a FIFO queue in decreasing order of connecting arc weights. The first vertex to be chosen is a source with the highest outbound arc weight, and the other sources are placed in the queue by decreasing order of their highest outbound arc weight. When a visited vertex has no successors or all of its successors are in a cluster, the next vertex is taken from the queue and a new cluster is created. As long as the size constraint M is respected, the visited vertices are placed in the cluster of the current neighbor. When the current cluster is full, a new cluster is created. The process ends when all vertices have been placed in a cluster.

Note that Algorithm A_1 works in polynomial time. Let $p_{\max}$ be the longest path in $P(G)$. Let $D \geq 1$ be the inter-cluster delay, and $d \geq 1$ the intra-cluster delay. Since $M \geq |p_{\max}|$, the vertices in the longest path can be grouped into the same cluster. Possible paths p intersecting $p_{\max}$ can include a subset of vertices placed into a different cluster. In the worst case, there exists a path p in a different cluster from the $p_{\max}$ cluster, such that $|p| = |p_{\max}|$, and p intersects $p_{\max}$. This case appears when there are enough longest paths p so that the sum of all of their

Algorithm 3 A_1 : hypergraph clustering algorithm

Require: H, M, D, d
Ensure: $\mathcal{C}$ a clustering of H

1: $r^* \leftarrow$ compute_criticality(H) ▷ Vertex criticality-ordered priority queue
2: $Q \leftarrow$ sort_by_criticality(r^*, H) ▷ Red source vertex criticality-ordered priority queue. We use $\max_u = \max_{v \in \Gamma^+(u)} r^*(u, v)$ to compare vertices
3: flag $\leftarrow$ [false]$^{|V|}$ ▷ Initialize flag array of visited vertices
4: $i \leftarrow 0$
5: $\mathcal{C} \leftarrow \emptyset$
6: **while** $Q \neq \emptyset$ **do**
7: $u \leftarrow Q$.dequeue() ▷ Get the most critical vertex
8: flag$[u] \leftarrow$ true
9: **if** $|\mathcal{C}_i| \geq M$ **then**
10: $i \leftarrow i + 1$
11: **end if**
12: $\mathcal{C}_i \leftarrow \mathcal{C}_i \cup \{u\}$
13: **for** $v \in \Gamma^+(u)$ **do**
14: **if** $\neg$flag$[v]$ **then**
15: insert_by_criticality_left(Q, v, r^*) ▷ v is necessarily a vertex of higher criticality, by propagation. When vertices have same criticality, insertion is performed to the left, to ensure grouping along the path
16: **end if**
17: **end for**
18: **end while**
19: **return** $\mathcal{C}$

vertices is greater than M. Since $M \geq |p_{\max}|$, $\forall p \neq p_{\max} \in P(G),\ |p \setminus p_{\max}| \leq M$, then $p \setminus p_{\max}$ is a cluster. The traversal order of the algorithm favors the clustering of long paths. In the worst case, its cost would equate $|p_{\max}| + D$, because G does not have a K_3 minor ($\iota(p) \leq 1$). Hence, algorithm A_1 produces a maximum path solution value bounded by: $|p_{\max}| \times d \leq \mathrm{Sol}_{A_1}(H) \leq |p_{\max}| \times d + D$ for all G, and returns the optimal solution. □

Lemma 4.3.4. *Let $G = (V, A)$ be a criticality DAG from a DAH, $P(G)$ the set of paths of G starting from a source and ending in a sink, such as $\forall p \in P(G),\ |P(G)| > 1,\ \iota(p) \leq 1$, and $p_{\max}$ be the longest path in $P(G)$. There exists a polynomial algorithm for the problem $CN{<}[1], M, \Delta{>}$ for $M{<}|p_{\max}|$.*

Algorithm 4 A_2 : hypergraph clustering algorithm

Require: H, M, D, d
Ensure: $\mathcal{C}$ a clustering of H
1: $\mathcal{C} \leftarrow A_1(H, M, D, d)$
2: $Q \leftarrow$ sort_by_criticality(r^*, H) ▷ Red source vertex criticality-ordered priority queue. We use $\max_u = \max_{v \in \Gamma^+(u)} r^*(u, v)$ to compare vertices
3: flag $\leftarrow$ [false]$^{|V|}$ ▷ Initialize flag array of visited vertices
4: **while** $Q \neq \emptyset$ **do**
5: $u \leftarrow Q$.dequeue() ▷ Get the most critical vertex
6: flag$[u] \leftarrow$ true
7: **for** $v \in \Gamma^+(u)$ **do**
8: **if** $\neg flag[v]$ **then**
9: insert_by_criticality_left(Q, v, r^*)
10: **end if**
11: **if** $(C(p_u) + C(p_v)) \times M \geq |p_u| + |p_v| + M$ **then**
12: merge_cluster$(p_u, p_v, \mathcal{C})$ ▷ Merge clusters along p_u and p_v to reduce the number of clusters by at least 1
13: **end if**
14: **end for**
15: **end while**
16: **return** C

Proof. Let Algorithm A_2 (displayed as Algorithm 4) be the algorithm operating as follows:

1. Execute Algorithm A_1 on H. Since $M < |p_{\max}|$, a number of paths intersecting $p_{\max}$ will be split into several clusters, leaving some leeway for cluster merging.

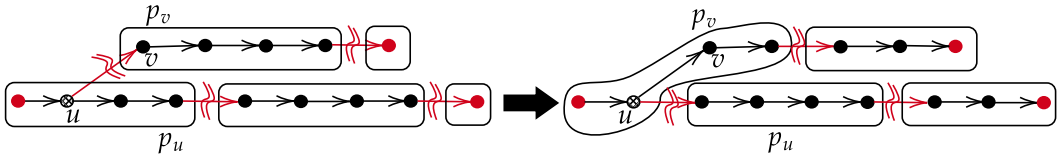


Figure 4.7: Example of cluster reduction for two neighbor paths p_u and p_v with $M = 4$. Two paths are considered to be neighbors iff they share at least one vertex. Algorithm A_1 produces the clustering in the left of the figure, in which: $(C(p_u) + C(p_v)) \times M \geq |p_u| + |p_v| + M = (3 + 3) \times 4 (=\mathbf{24}) \geq 9 + 7 + 4 (=\mathbf{20})$. The procedure `merge_cluster` transform the cluster in the left of the figure to a clustering in the right, and decreases by 1 the number of clusters along p_v.

2. In a visit order similar to that of Algorithm A_1, for each vertex v, if the leeway for the cluster including v and that of an adjacent cluster enables merging, include the adjacent cluster into the cluster comprising v.

Let $D \geq 1$ be the inter-cluster delay and $d \geq 1$ the intra-cluster delay. As $M < |p_{\max}|$, the longest paths will create at most $\lfloor \frac{|p_{\max}|}{M} \rfloor$ clusters. Possible paths intersecting $p_{\max}$ can have part of their vertices contained in clusters that do not contain vertices of $p_{\max}$. In the worst case, there is a path p such that $|p| = |p_{\max}|$ with the vertices of p in a different cluster from the clusters of $p_{\max}$, save for at least one vertex, since p intersects $p_{\max}$. For this case, the cost would be equal to $\lfloor \frac{|p_{\max}|}{M} \rfloor \times D + D$. Let $C(p)$ be the number of clusters along a path p such that $|p| = |p_{\max}|$; p intersects $p_{\max}$ and $(C(p) + C(p_{\max})) \times M \geq |p_{\max}| + |p| + M$. In this case, we can reduce the number of clusters by moving $|p| - \lfloor \frac{|p|}{M} \rfloor \times M$ vertices from p to a neighboring cluster. The number of clusters along $p_{\max}$ will not change, but the number of cluster along p can decreased by one. The second part of the algorithm corresponds to this case, an example of which can be found in Figure 4.7. Note that we only move what is necessary to allow the reduction of the number of clusters for possible other paths close to $p_{\max}$.

After each move, the number of vertex moves is decremented. If this value reaches 0, reducing the number of clusters without degrading previous optimizations is no longer possible. Note that Algorithm A_2 works in polynomial time and can be bounded by: $\lfloor \frac{|p_{\max}|}{M} \rfloor \times D \leq Sol_{A_2}(H) \leq \lfloor \frac{|p_{\max}|}{M} \rfloor \times D + D$, and returns the optimal solution for $\iota(p) \leq 1, \forall p \in H$. □

4.3.3 NP-Completeness

In this subsection, we extend the proof of NP-Completeness of the CN problem to red-black hypergraphs. Z. Donovan presented a reduction of the integer set

partitioning problem to the CN problem for DAGs [59]. Based on this strategy, we propose a similar reduction based on the structure of red-black hypergraphs.

Theorem 4.3.5. *Let H be a DAH and $G = (V, A')$ be its corresponding criticality DAG, such that $\forall p \in P(H)$, $\iota(p) \geq 2$. Unless $P = NP$, there is no polynomial algorithm to solve the problem $CN{<}[1], M, \Delta{>}$.*

Proof. Let us define and extend to red-black hypergraphs the definition of the decision version of the CN problem previously defined by Z. Donovan, as CN_{dec}:

> *Given a red-black Hypergraph $H = (V, A)$, with a vertex weighting function $w : V \to \mathbb{R}^+$, a delay function $d : V \to \mathbb{R}^+$, a maximum degree Δ, a constant D, and a cluster capacity M, and a positive integer d^*, decide whether we can partition V into clusters such that:*
>
> (i) *the weight of each cluster is bounded by M ;*
>
> (ii) *the maximum delay-length of any red-red path of H is at most d^*.*

The CN_{dec} problem belongs to the NP class because it is possible to find the critical path in a red-black hypergraph with a polynomial time algorithm [49]. Since there is no cycle within red-red paths, we can use an algorithm based on topological sorting.

As presented in the work of Z. Donovan, we will use the same reduction strategy from the red-black hypergraph structure, *i.e.*, a reduction from the PARTITION problem. Let PARTITION be the problem defined as follows: given a set of integers $I = \{i_1, ..., i_n\}$, the goal is to find a partition of I into two subsets $I_1 \subset I$ and $I_2 \subset I$, such that:

$$\sum_{i \in I_1} i_1 = \sum_{i \in I_2} i_2 \ . \tag{4.8}$$

Let us create an instance of CN_{dec} as shown in Figure 4.8. It contains a source red vertex connected to a sink red vertex through n vertices with weights $i_1, ..., i_n$. The two red vertices have weight $\frac{\sum_{i \in I} i}{2} = B$. We assume that $\sum_{i \in I} i$ is even; otherwise, the problem is trivially unsolvable. We set the parameters of CN_{dec} to $d = 0$ and $D > 0$. For each vertex v, the delay of v is 0. The cluster capacity is set to $2 \times B$, and we set $d^* = D$ to ensure only one cut along all paths.

Note that the translation of this problem to an instance X^P for the PARTITION problem can be done in polynomial time. To complete the proof, we will show that an instance X^P of PARTITION is valid if and only if its analogous instance X^{CN} is valid for the problem CN_{dec}.

Suppose X^P is an instance such that PARTITION(X^P) = True. Then, there exists a partition of I into I_1 and I_2 such that $\sum_{i \in I_1} i_1 = \sum_{i \in I_2} i_2 = \frac{\sum_{i \in I} i}{2} = B$.

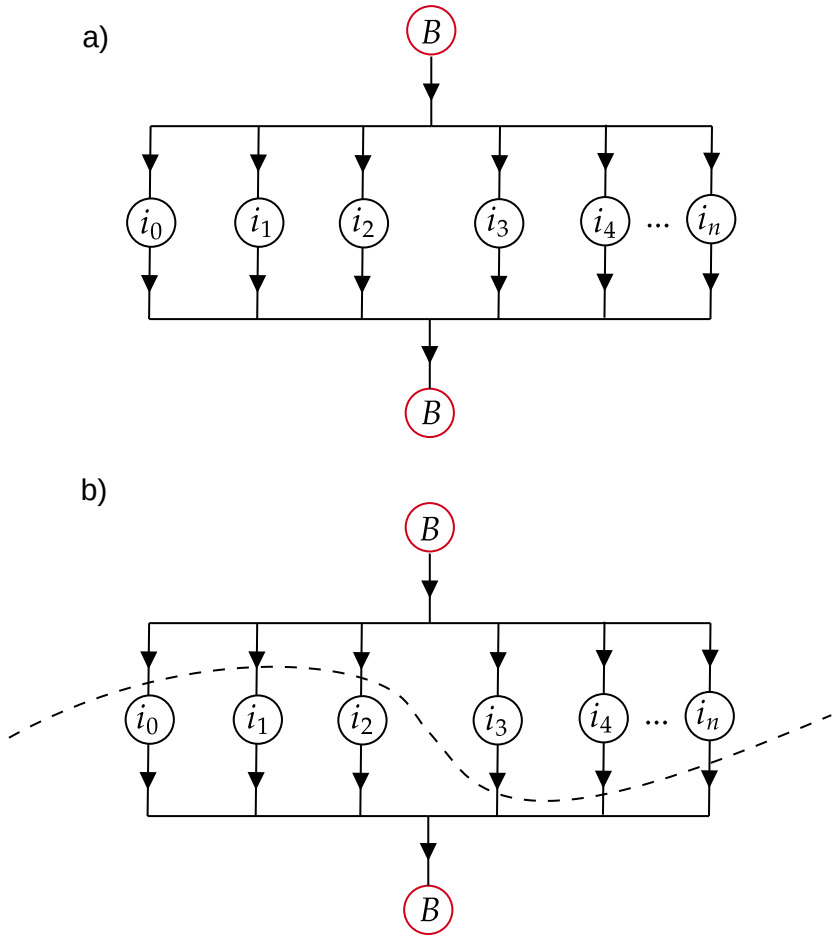


Figure 4.8: a) Example of a partition problem instance for the CN_{dec} problem. b) Example of a valid solution for the CN_{dec} problem with $d^* = D$, *i.e.*, only one cut per path is allowed.

Suppose we group the vertices of I_1 with the red source vertex and those of I_2 with the red sink vertex. In that case, we obtain two clusters of capacity $2 \times B$, the maximum capacity associated with each cluster. Note that the value of the critical path is equal to $D = d^*$. Hence, X^{CN} is also a valid instance of the CN_{dec} problem.

Now, we will demonstrate the other way, *i.e.*, if X^{CN} is a valid instance for the problem CN_{dec}, then the analogous instance X^P is valid for the PARTITION problem. If X^{CN} is a valid instance for the problem CN_{dec}, then a partition of vertices exists such that the critical path is equal to $d^* = D$. Note that the red vertices cannot form a cluster. Otherwise, the critical path would be cut twice and its cost would equal $2 \times D > d^*$. Without loss of generality, let C_1 be the cluster with the source vertex and C_2 be the cluster with the sink vertex, and $W(C_1)$ and $W(C_2)$ be the sum of the weights of the vertices in clusters 1 and 2. We have:

$$\begin{cases} W(C_1) = W(v^R) + \sum_{v \in C_1 \setminus \{v^R\}} = B + B = 2 \times B, \\ W(C_2) = W(v^R) + \sum_{v \in C_2 \setminus \{v^R\}} = B + B = 2 \times B. \end{cases} \tag{4.9}$$

Therefore:

$$\sum_{v \in C_1 \setminus \{v^R\}} = \sum_{v \in C_2 \setminus \{v^R\}} = B = \frac{\sum_{i \in I} i}{2} \ . \tag{4.10}$$

An illustration can be found in Figure 4.8.

Consequently, we have a valid partition for I, and X^P is a valid instance of PARTITION.

□

The proof of Theorem 4.3.5, originally written for DAGs, can be found in [59]. Indeed, $\iota(p) \geq 2$ is a necessary condition to construct the reduction to the partition set problem.

4.3.4 Conclusion

In this section, the iota metric ι has been introduced. This metric models a connectivity cost for paths. Paths are an essential aspect for a critical path clustering problem. We showed that when paths are strongly interconnected, it is more complicated to cluster critical paths. However, a polynomial-time algorithm is introduced for the case $\iota \leq 1$. In practice, this algorithm cannot be applied on circuits with $\iota > 1$ without a pre-processing modeling them by circuits with $\iota \leq 1$.

We showed again that the problem is NP-Complete for $\iota > 1$, and adapted this proof to our red-black hypergraph model. This result implies that circuits with $\iota \leq 1$ cannot exactly model circuits with $\iota > 1$.

4.4 A parameterized M-approximation algorithm for red-black hypergraph clustering

Since the clustering problem is NP-hard and there is no approximation algorithm with a constant factor in the general case, approximation algorithms have been proposed to provide acceptable solutions in reasonable time, such as the parameterized $M^2 + M$ approximation algorithm presented by Z. Donovan *et al.* [62]. We propose an improved approximate algorithm of the latter, based on binary search.

4.4.1 Binary Search Clustering (BSC)

Let $H = (V, A)$ be a DAH, and $p_{\max}$ its critical path. Let ϕ be a feasible minimum cost, $\phi \in [|p_{\max}| \times d, |A| \times D]$, with D the inter-cluster delay and d the intra-cluster delay. Given a fixed value ϕ, we can define a cut capacity for each pair of vertices (u, v) as:

$$\text{cut\_cap}(u, v) \;=\; \frac{\max(0, \phi - r^*(u, v))}{D} \quad . \tag{4.11}$$

Suppose the cut capacity between two vertices u and v equals zero. Then, u and v should be placed in the same cluster. As the size of the cluster is constrained by the parameter M, it is possible to know whether some ϕ is unfeasible, by exceeding some cluster size.

Lemma 4.4.1. *The binary search clustering runs in $O(m \cdot log_2(m))$, with m being the number of arcs in the associated critical DAG.*

Proof. Algorithm 5 contains a `while` loop that will perform at most $\log_2(m)$ iterations. Lines 1 and 2 of the algorithm define the lower and upper bounds of the binary search. Even if the hypergraph is a path, *i.e.*, if the lower bound is equal to m and the upper bound is equal to m^2, the number of iterations of the `while` loop will be in $O(\log_2(m^2))$, which does not change the order of complexity.

Line 6 calls a procedure that works in $O(m)$ time. Indeed, the procedure computes the cut capacity of every arc and merges every pair of vertices with a cut capacity equal to zero. In line 10, to cluster the remaining unclustered vertices connected by arcs with non-negative cutting capacity, BSC calls the $O(m)$ algorithm A_1, presented as Algorithm 3 in Section 4.3.3. Hence, the time complexity of this algorithm is in $O(m \cdot \log_2(m))$. □

The algorithm presented by Z. Donovan [59] has a complexity in $O(2^{\Delta \cdot M} + |V|^{O(1)})$ time. For a sufficiently large M, this algorithm can become impractical. The BSC algorithm has a complexity time in $O(m \cdot \log_2(m))$, which is better in practice. The typical sparsity of circuits is relatively low, m remains small with respect to $|V|$.

Algorithm 5 Binary Search Clustering

Require: H, M, D, d
Ensure: $\mathcal{C}$ a clustering of H
1: $\overline{\phi} \leftarrow |A| \times D$
2: $\underline{\phi} \leftarrow |p_{\max}| \times d,\ p_{\max} \in H$
3: **while** $\overline{\phi} > \underline{\phi}$ **do**
4: $\quad \phi_{target} \leftarrow \frac{\underline{\phi}+\overline{\phi}}{2}$
5: $\quad \triangleright$ Compute the cut capacity for every pair $(uv) = \frac{\max(0, r^*(u,v) - \phi_{\text{target}})}{D}$ and for all pairs with cut capacity equal to zero, place u and v into the same cluster
6: $\quad \mathcal{C} \leftarrow$fusion_cut_cap$(H, \phi_{\text{target}}, \text{max_size})$
7: $\quad$ **if** $\max_{c \in \mathcal{C}} |c| \leq M$ **then**
8: $\quad\quad \overline{\phi} \leftarrow \phi_{\text{target}}$
9: $\quad$ **else**
10: $\quad\quad \underline{\phi} \leftarrow \phi_{\text{target}}$
11: $\quad$ **end if**
12: **end while**
13: `Call` A_1 `to cluster yet unclustered vertices`
14: **return** $\mathcal{C}$

Theorem 4.4.2. *The binary search clustering is an M-approximation algorithm for $CN<[w], M, \Delta>$ when $|p_{\max}| \times d > D$, $D \gg d$ and $\frac{D}{d} \leq M$, with $p_{\max}$ the critical path, d an intra-cluster delay, D an inter-cluster delay and M the maximum size of clusters.*

Proof. Let $H = (V, A)$ be a DAH, and $G = (V, A)$ be its corresponding criticality DAG. Let $|p_{\max}|$ be the longest path in H. As each vertex has a weight w, we will consider that $w = 1$. Let $\text{Sol}^*(H)$ be the optimal solution for a vertex-set clustering of H, an intra-cluster delay d, and an inter-cluster delay D, such that $D \gg d$ and $\frac{D}{d} \leq M$.

$$\text{Sol}^*(H) \geq \left(\left\lceil \frac{|p_{\max}|}{M} \right\rceil - 1 \right) \times D + \left(|p_{\max}| - 1 - \left(\left\lceil \frac{|p_{\max}|}{M} \right\rceil - 1 \right) \right) \times d \ . \quad (4.12)$$

Let $p_{\max}$ the critical path; we suppose $|p_{\max}| \times d > D$. Hence, we obtain:

$$|p_{\max}| \times d - D > 0 \ . \quad (4.13)$$

In many cases, the propagation time of a circuit's critical path is longer than the time it takes to transfer a signal from one FPGA to another. However, there are circuits for which this is not true, although they are very few. Therefore, this proof applies only to circuits that satisfy Equation 4.4.1.

The BSC algorithm groups vertices using a direct approach based on cut capacity. This makes it more practical than a recursive coupling approach. Let $\mathrm{Sol}_{\mathrm{bsc}}(H)$ be the solution produced by our BSC algorithm, presented as Algorithm 5. It can be bounded by the worst solution. A worst-case solution is one in which each vertex forms a cluster. Hence, we have:

$$\mathrm{Sol}_{\mathrm{bsc}} \leq (|p_{\max}| - 1) \times D \ . \tag{4.14}$$

Then, the approximation ratio is defined by:

$$\frac{\mathrm{Sol}_{\mathrm{bsc}}(H)}{\mathrm{Sol}^*(H)} \leq \frac{(|p_{\max}| - 1) \times D}{\left(\left\lceil \frac{|p_{\max}|}{M} \right\rceil - 1\right) \times D + \left(|p_{\max}| - 1 - \left(\left\lceil \frac{|p_{\max}|}{M} \right\rceil - 1\right)\right) \times d} \ . \tag{4.15}$$

Let us calculate the approximation ratio for $|p_{\max}| > M$ and $|p_{\max}| \leq M$. In the case when $|p_{\max}| > M$:

$$\frac{\mathrm{Sol}_{\mathrm{bsc}}(H)}{\mathrm{Sol}^*(H)} \leq \frac{(|p_{\max}| - 1) \times D}{\left(\left\lceil \frac{|p_{\max}|}{M} \right\rceil - 1\right) \times D + \left(|p_{\max}| - 1 - \left(\left\lceil \frac{|p_{\max}|}{M} \right\rceil - 1\right)\right) \times d} \ .$$

As $|p_{\max}| > M$, we have:

$$\left\lceil \frac{|p_{\max}|}{M} \right\rceil = \frac{|p_{\max}| + (M + r)}{M} \ . \tag{4.16}$$

By applying Equation 4.16, we obtain:

$$\begin{aligned} \frac{\mathrm{Sol}_{\mathrm{bsc}}(H)}{\mathrm{Sol}^*(H)} &\leq \frac{(|p_{\max}|-1)\times D}{\frac{|p_{\max}|+(M-r)-M}{M}\times D + \frac{M\times|p_{\max}|-|p_{\max}|-(M-r)}{M}\times d} \\ &= M\frac{(|p_{\max}|-1)\times D}{(|p_{\max}|-r)\times D+(M-1)\times|p_{\max}|\times d-(M-r)\times d} \ . \end{aligned}$$

By applying Equation 4.4.1, we obtain:

$$\begin{aligned} \frac{\mathrm{Sol}_{\mathrm{bsc}}(H)}{\mathrm{Sol}^*(H)} &\leq M\frac{(|p_{\max}|-1)\times D}{(|p_{\max}|-r)\times D+(M-1)\times D-(M-r)\times d} \\ &= M\frac{(|p_{\max}|-1)\times D}{(|p_{\max}|-r+M-1)\times D-(M-r)\times d} \ . \end{aligned}$$

Let us study the positivity of the expression $DM - Dr - (M - r)d$:

$$\begin{aligned} D(M-r) &> (M-r)d, \ (D \gg d) \\ &= DM - Dr > (M-r)d \ . \end{aligned}$$

Hence, we obtain:

$$DM - Dr > (M - r)d = DM - Dr - (M - r)d > 0 \ . \tag{4.17}$$

By applying Equation 4.17, we obtain:

$$\begin{aligned} \frac{\mathrm{Sol}_{\mathrm{bsc}}(H)}{\mathrm{Sol}^*(H)} &\leq M\frac{(|p_{\max}|-1)\times D+DM-Dr-(M-r)\times d}{(|p_{\max}|-r+M-1)\times D-(M-r)\times d} \\ &= M\frac{(|p_{\max}|+M-r-1)\times D-(M-r)\times d}{(|p_{\max}|+M-r-1)\times D-(M-r)\times d} \\ &= M \ . \end{aligned}$$

In the case when $|p_{\max}| \leq M$:

$$\frac{\mathrm{Sol}_{\mathrm{bsc}}(H)}{\mathrm{Sol}^*(H)} \leq \frac{(|p_{\max}|-1)\times D}{\left(\left\lceil\frac{|p_{\max}|}{M}\right\rceil-1\right)\times D+\left(|p_{\max}|-1-\left(\left\lceil\frac{|p_{\max}|}{M}\right\rceil-1\right)\right)\times d} \ .$$

As $|p_{\max}| \leq M$, we have:

$$\left\lceil\frac{|p_{\max}|}{M}\right\rceil = 1 \ . \tag{4.18}$$

By applying Equation 4.18, we obtain:

$$\frac{\mathrm{Sol}_{\mathrm{bsc}}(H)}{\mathrm{Sol}^*(H)} = \frac{(|p_{\max}|-1)\times D}{(|p_{\max}|-1)\times d} \ .$$

Since $\frac{D}{d} \leq M$, we obtain:

$$\frac{\mathrm{Sol}_{\mathrm{bsc}}(H)}{\mathrm{Sol}^*(H)} = \frac{(|p_{\max}|-1)\times D}{(|p_{\max}|-1)\times d} = \frac{D}{d} \leq M \ .$$

Hence, the parameterized approximation ratio is M for $CN{<}[w], M, \Delta{>}$ under the condition specified in Theorem 4.4.2. In the general case, the ratio remains $M^2 + M$. □

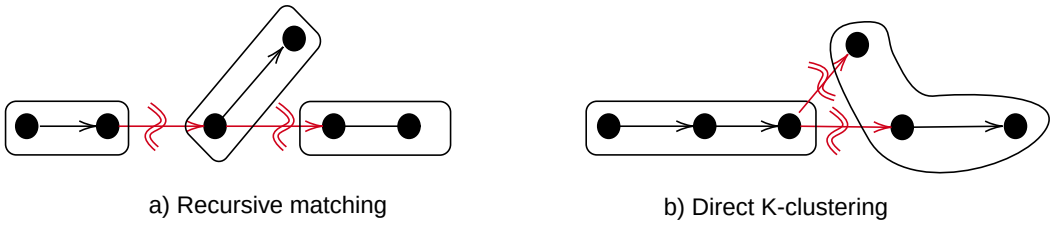


Figure 4.9: This figure presents the effects of recursive matching vs. direct k-way clustering. On the left is a solution produced by a recursive matching algorithm for clustering with $M = 3$. On the right is the result of a direct clustering. This example evidence that direct clustering produces less clusters, hence possibly less cuts, than the recursive matching approach.

4.4.2 Heavy-edge matching

The heavy-edge matching (HEM) approach for graph clustering, presented by G. Karypis *et al.* [108], is widely used in state-of-the-art (hyper)graph partitioning tools [110, 149] and yields efficient results in many cases. The advantage of this algorithm is that, in the unconstrained case, it almost halves the size of the instance during each of the first stages of the multilevel framework, which makes its complexity more interesting than that of our Algorithm 5. However, we will show in this subsection that HEM, as well as other algorithms dedicated to 2-matching introduced by Z. Donovan *et al.* [59, 61], do not capture path topology adequately. An example is presented in Figure 4.9 for a clustering with $M > 2$. We will also show that HEM, applied to some criticality DAG (weighted with the r^* scheme), yields an approximation ratio of 2 for the $CN{<}[1], 2, \Delta{>}$ problem. This algorithm differs from the two algorithms presented by Z. Donovan *et al.* [59, 61]: one of them looks for a dominant matching, and otherwise returns an arbitrary clustering, while the other is based on a linear programming rounding algorithm.

In the example shown in Figure 4.9, the recursive methods will match vertices only once and cannot match them at the next level, because new vertices have a weight equal to 2. A direct clustering algorithm like our Algorithm 5 will produce in this case a result as good as recursive matching methods. This suggests that a direct clustering algorithm will be more interesting than a recursive coupling algorithm when M is large.

Theorem 4.4.3. *Let H be a DAH and $G = (V, A)$ be its corresponding criticality DAG. The HEM algorithm applied to the DAG for $CN{<}[1], 2, \Delta{>}$ is a 2-approximation algorithm.*

Proof. Let d be the intra-cluster delay and D be the inter-cluster delay, such that

$D \gg d$. Let $p_{\max}$ be the critical path, with $|p_{\max}| \times d > D$. Hence:

$$|p_{\max}| \times d - D > 0 \ . \tag{4.19}$$

Let $\mathrm{Sol}^*(H)$ be the optimal solution for a vertex set clustering of H. In the best case, for a cluster size bounded by 2, the critical path will be coupled $\frac{|p_{\max}|}{2}$ times, which will yield the following lower bound for $\mathrm{Sol}j(H)$:

$$\mathrm{Sol}^*(H) \geq \left(\left\lceil \frac{|p_{\max}|}{2} \right\rceil - 1\right) \times D + \left(|p_{\max}| - 1 - \left(\left\lceil \frac{|p_{\max}|}{2} \right\rceil - 1\right)\right) \times d \ . \tag{4.20}$$

Let $\mathrm{Sol}_{\mathrm{HEM}}(H)$ be the solution produced by the HEM scheme on our proposed criticality DAG model. It can be bounded by the worst possible solution, in which every vertex forms a cluster. Hence:

$$\mathrm{Sol}_{\mathrm{HEM}}(H) \leq (|p_{\max}| - 1) \times D \ . \tag{4.21}$$

Then, the approximation ratio is defined by:

$$\frac{\mathrm{Sol}_{\mathrm{HEM}}(H)}{\mathrm{Sol}^*(H)} \leq \frac{(|p_{\max}| - 1) \times D}{\left(\left\lceil \frac{|p_{\max}|}{2} \right\rceil - 1\right) \times D + \left(|p_{\max}| - 1 - \left(\left\lceil \frac{|p_{\max}|}{2} \right\rceil - 1\right)\right) \times d} \ . \tag{4.22}$$

Let us calculate the approximation ratio for the even and odd cases of $|p_{\max}|$. When $|p_{\max}|$ is even:

$$\begin{aligned}
\frac{\mathrm{Sol}_{\mathrm{HEM}}(H)}{\mathrm{Sol}^*(H)} &\leq \frac{(|p_{\max}|-1)\times D}{\left(\left\lceil \frac{|p_{\max}|}{2} \right\rceil -1\right)\times D + \left(|p_{\max}| - 1 - \left(\left\lceil \frac{|p_{\max}|}{2} \right\rceil -1\right)\right)\times d} \\
&= \frac{(|p_{\max}|-1)\times D}{\frac{|p_{\max}|}{2}\times D - \frac{2D}{2} + \frac{|p_{\max}|}{2}\times d + \frac{2d}{2}} \\
&= 2 \times \frac{|p_{\max}|D - D + D - D}{|p_{\max}|D - 2D + |p_{\max}|d + 2d} \ .
\end{aligned}$$

By applying Equation 4.19, we obtain:

$$\begin{aligned}
\frac{\mathrm{Sol}_{\mathrm{HEM}}(H)}{\mathrm{Sol}^*(H)} &\leq 2 \times \frac{|p_{\max}|D - 2D + |p_{\max}|d + 2d - D}{|p_{\max}|D - 2D + |p_{\max}|d + 2d} + \frac{2D}{|p_{\max}|D - 2D + |p_{\max}|d + 2d} \\
&= 2 \times \frac{|p_{\max}|D - 2D + |p_{\max}|d + 2d}{|p_{\max}|D - 2D + |p_{\max}|d + 2d} \\
&\frac{\mathrm{Sol}_{\mathrm{HEM}}(H)}{\mathrm{Sol}^*(H)} \leq 2 \ .
\end{aligned}$$

When $|p_{\max}|$ is odd:

$$\begin{aligned}
\frac{\mathrm{Sol}_{\mathrm{HEM}}(H)}{\mathrm{Sol}^*(H)} &\leq \frac{(|p_{\max}|-1)\times D}{\left(\left\lceil\frac{|p_{\max}|}{2}\right\rceil-1\right)\times D+\left(|p_{\max}|-1-\left(\left\lceil\frac{|p_{\max}|}{2}\right\rceil-1\right)\right)\times d} \\
&= \frac{(|p_{\max}|-1)\times D}{\left(\frac{|p_{\max}|+1}{2}-1\right)\times D+\left(|p_{\max}|-1-\left(\frac{|p_{\max}|+1}{2}-1\right)\right)\times d} \\
&= \frac{(|p_{\max}|-1)\times D}{\left(\frac{|p_{\max}|}{2}+\frac{1}{2}-1\right)\times D+\left(|p_{\max}|-1-\frac{|p_{\max}|}{2}-\frac{1}{2}+1\right)\times d} \\
&= \frac{(|p_{\max}|-1)\times D}{\left(\frac{|p_{\max}|}{2}-\frac{1}{2}\right)\times D+\left(\frac{|p_{\max}|}{2}-\frac{1}{2}\right)\times d} \\
&= 2\times\frac{|p_{\max}|D-D}{|p_{\max}|D-D+|p_{\max}|d-d} \ .
\end{aligned}$$

Note that the critical path contains at least 2 vertices, such that $|p_{\max}| \geq 2$. Hence:

$$\begin{aligned}
\frac{\mathrm{Sol}_{\mathrm{HEM}}(H)}{\mathrm{Sol}^*(H)} &\leq 2\times\frac{|p_{\max}|D-D+|p_{\max}|d-d}{|p_{\max}|D-D+|p_{\max}|d-d} = 2 \\
\frac{\mathrm{Sol}_{\mathrm{HEM}}(H)}{\mathrm{Sol}^*(H)} &\leq 2 \ .
\end{aligned}$$

Hence, the HEM algorithm, applied to our local critical path model represented by the corresponding DAG, has an approximation ratio of 2 for the $CN{<}[1], 2, \Delta{>}$ problem. □

For $M = 2$, the matching algorithm behaves in the same way as the BSC algorithm with our r^* weighting. They have the same approximation ratio for $M = 2$.

4.4.3 Conclusion

In this section, a parameterized approximation algorithm is presented. Its parameter is M, the cluster capacity. We prove that BSC, presented as Algorithm 5 has an approximation ratio of M. We also prove that HEM, used with our r^* weighting scheme, has a 2-approximation ratio.

The BSC Algorithm 5 improves existing parameterized approximation ratios of algorithms for clustering with path-length minimization.

4.5 Experimental Results

To validate our models and algorithms, we have performed experiments on benchmarks of the logic circuits presented in Chapter 3, which consist of acyclic combinatorial blocks bounded by their input and output registers. Every combinatorial block can therefore be modeled as a DAH. Their computation time is conditioned by their critical path, defined as the longest path between two registers (*i.e.*, two red vertices).

Remember that we want to minimize the number of cuts on the critical path. In fact, in our problem, a cut on a path means an additional delay in the path cost. Thus, the compared algorithms aim to group the red-black hypergraphs by minimizing path length, *i.e.*, the maximum path cost $p_{\max}$. Since the execution time and the number of clusters are important indicators, we measure and compare them. Note that clustering minimizing the number of clusters refers to the bin-packing problem, which is known to be NP-hard.

To compare BSC and HEM, we measured the ratio of average BSC results to average HEM results for M values ranging from 2 to 4096. Each algorithm was run 10 times for each circuit and for each size. The average of these 10 runs was used to calculate the averages for all instances per size. More details about the results, such as the standard deviation, can be found in Appendix A.1.

We measured the degradation of the critical path produced by algorithm $\mathcal{A}$ for each instance I, calculated by:

$$(\mathrm{Sol}_{\mathcal{A}}(I) - p^I_{\max})/p^I_{\max} \ . \tag{4.23}$$

The results in Figure 4.10 show that our BSC clustering algorithm, applied to circuit hypergraph, outperforms the HEM algorithm for critical path degradation. It can be shown that HEM points are more on the left side than BSC plots. This relies on the fact that HEM takes less execution time than BSC. However, the execution time of HEM increases along with cluster size, that is, some HEM points moves from left to right. Indeed, as we increase the size of the clusters, we notice that HEM makes more recursive calls. Even if these recursive calls are executed on reduced hypergraphs, this increases run time. As a result, the complexity of HEM can be described by an additional factor of $\log_2(M)$, while the time complexity of the BSC algorithm admits a time complexity that depends only on the number of hyperedges. In practice, however, we find that the execution time of the BSC algorithm varies slightly as a function of M during the grouping phase, since this phase differs for each M. For BSC, however, these variations remain negligible, which explains why the plots of BSC do not change on abscissa of Figure 4.10 for each cluster size M.

The results in Figure 4.11 show that each BSC curve is under the HEM curve, that is, our BSC clustering algorithm produces fewer clusters compared with the

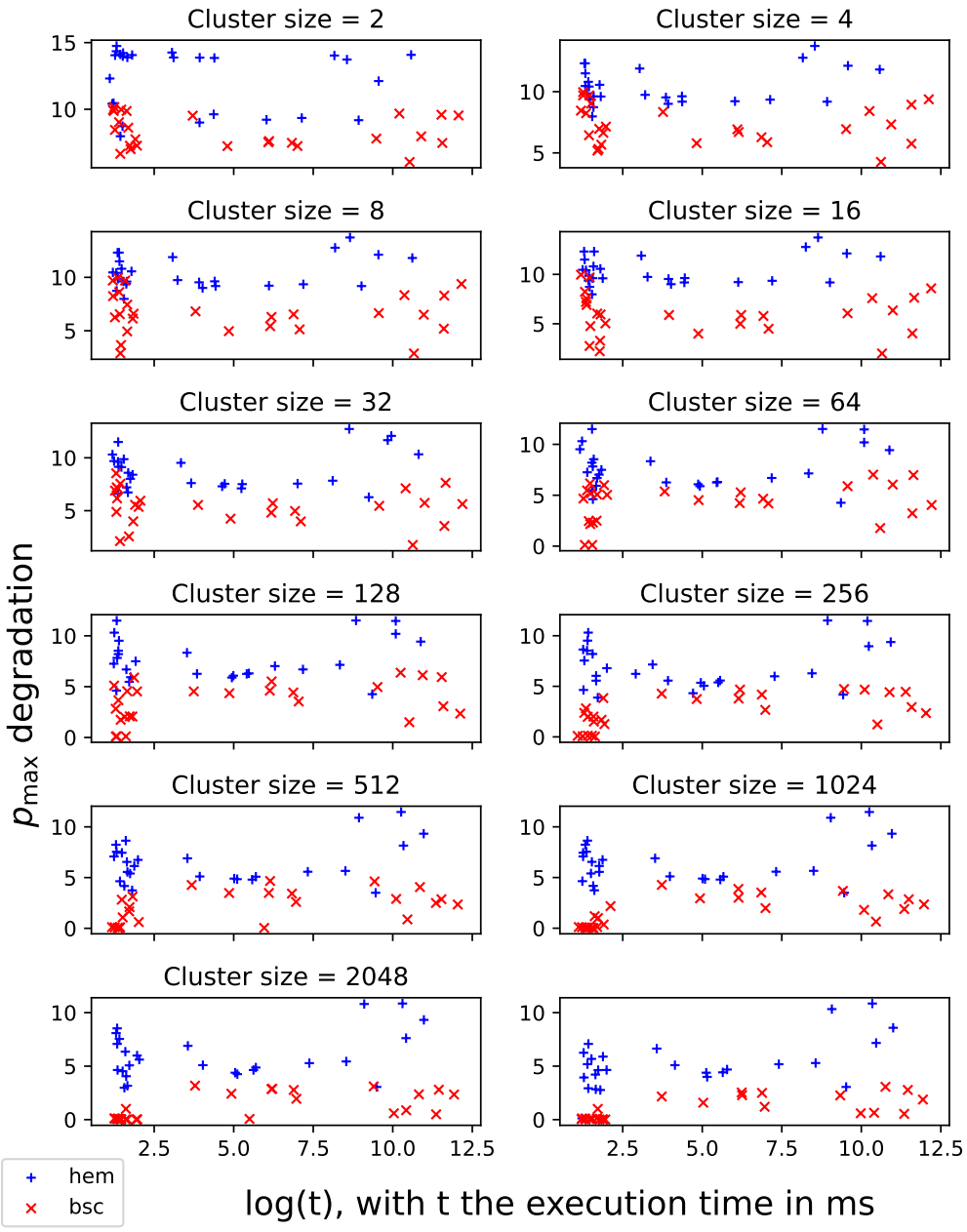


Figure 4.10: Results of BSC and HEM on each circuit (plot) for M values ranging from 2 to 4096. Each " + " plot is a clustering result calculated by the HEM algorithm and each " × " plot is for BSC. Each plot is defined by the degradation of the critical path (ordinate) as a function of its logarithmic execution time (abscissa). As shown in all sub-figures, the "×" plot are positioned below the "+" plot, which indicates a lower critical path degradation for BSC. In addition, each plot positioned to the left is based on a lower execution time. For two circuits, BSC takes more time than HEM.

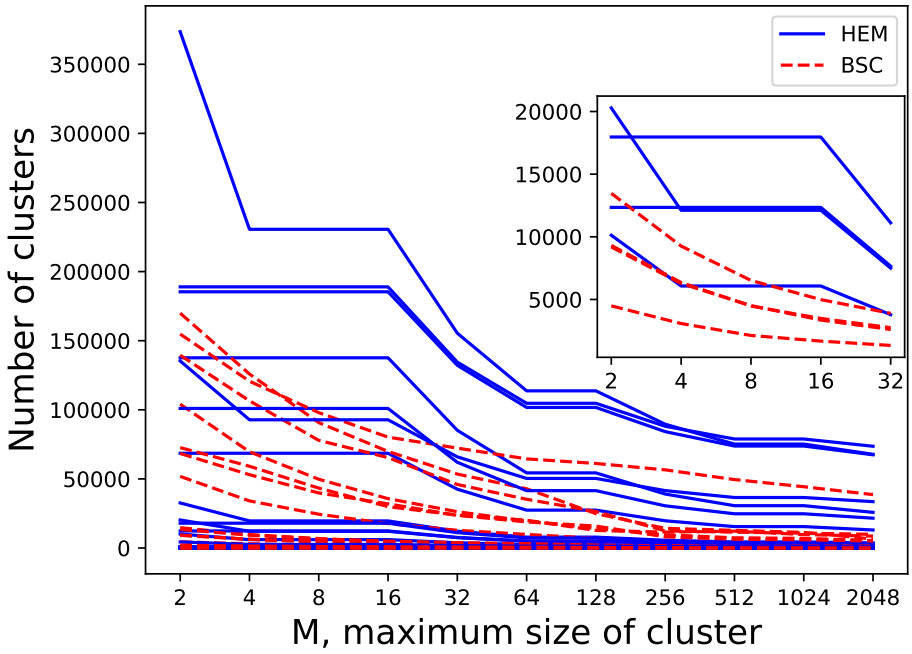


Figure 4.11: Comparison between the number of clusters produced by BSC and HEM on a subset of the largest circuits, for M values ranging from 2 to 4096. The subset is composed of Chipyard, TITAN and B14-22, all instances with $|V| > 10000$. Each plain line corresponds to the number of clusters produced by HEM, and hatched lines to the number of clusters produced by BSC. Results show that BSC produces fewer clusters than HEM. The subfigure is a zoom of B14, B20, B21, and B22, for M values ranging from 2 to 32.

HEM algorithm. This can be explained by the fact that BSC directly groups sets of related vertices and applies a final clustering step that tends to reduce the number of clusters. In contrast, the HEM algorithm recursively groups vertices in pairs, which can more easily lead to situations where there are several adjacent clusters of size $M/2 + 1$ that cannot be merged.

4.6 Conclusion

In this chapter, we studied the combinatorial circuit clustering problem for delay minimization (CN). We presented a brief state of the art in Section 4.1. The central aspect of clustering algorithms is to select vertices to merge. Thus, the key is to establish a good attractiveness metric between the vertices that best suits the objective. In Section 4.2 we presented the r^* weighting scheme, specifically designed for this purpose.

Since the problem is NP-hard in the general case, polynomial algorithms have been presented in Section 4.3 for a specific class of red-black hypergraphs the criticality DAG representation of which is a tree. A new M-approximation algorithm that runs in $O(m \cdot \log_2(m))$ time, with M being the maximum size of clusters and m the number of hyperarcs, is introduced in Section 4.4.

Section 4.5 shows a comparison between the classic HEM algorithm, improved with our weighting scheme r^*, and our BSC algorithm also using r^*. Experimental results show that BSC improves delay length by 20% to 50% on average, on all circuits, for many cluster sizes.

Chapter 5

Initial partitioning

The second stage of the multilevel partitioning scheme is *initial partitioning*. It consists in applying a partitioning method to a coarsened graph or hypergraph.

This chapter presents a brief state of the art on greedy partitioning algorithms in the context of red-black (circuit) hypergraph partitioning. Most of the algorithms presented in this state of the art have been developed to minimize different costs, particularly the cut size. As we have shown in previous chapters, cut minimization cannot minimize path-cost; hence, in Section 5.2, we designed dedicated greedy algorithms based on traversal algorithms. Section 5.3 introduces an integer programming approach based on cut and path-length minimization.

In the context of circuit partitioning, a multilevel scheme typically uses such algorithms for its initial partitioning phase. This chapter is based on some of our published works [160, 161].

5.1 Graph and hypergraph partitioning

The problem of partitioning graphs and hypergraphs is known to be NP-hard unless $P = NP$ [78, 125]. We have proved in Theorem 4.3.5 that the problem of partitioning a red-black hypergraph to minimize critical path degradation is also NP-hard. Consequently, to provide good solutions in reasonable time, the scientific community has developed approximated algorithms with polynomial time complexity. One of them is the multilevel scheme introduced in Chapter 2. The multilevel scheme reduces first the size of partitioning problem before computing an initial partition, followed by a refinement algorithm.

The initial partitioning phase involves algorithms, which mostly are greedy algorithms, that aim to compute an initial partition.

Most partitioning algorithms, are designed to minimize the size of the cut. However, various works [4, 127] have shown that an algorithm that minimizes the cut cannot minimize the path cost. Furthermore, most of these algorithms, including those presented in this chapter, do not take into account the target topology during the partitioning procedure.

The first approach studied in this chapter refers to adapting traversal algorithms which can provide contiguous partitions, *i.e.*, grouping neighbor vertices in the same part while satisfying the capacity constraint. Contiguous partitions are interesting in red-black hypergraph partitioning, particularly along critical paths, because they favor grouping neighbor vertices in the same part. However, one has to modify the classical traversal algorithm to define different strategies of contiguity that take into account the criticality of the vertices. Indeed, vertex criticality is a relevant metric for evidencing the potential impact of a vertex on the cost function.

In the multilevel scheme, initial partitioning algorithms are applied to the

smallest hypergraph, obtained by multiple clustering steps. Since these hypergraphs can be arbitrarily small, we studied the use of an integer programming approach as initial partitioning. This integer program is defined as a multi-objective for cut and path-cost minimization.

Other works are based on label propagation algorithms. These algorithms were first developed to detect communities in large graphs [156, 194]. H. Meyerhenke *et al.* [133] presented a label propagation algorithm applied to partitioning. The principle of this approach is to define a label for each vertex that represents its group. The group can be a part or a cluster. In the initial stage of the algorithm, each vertex has a label $L(u) = u$. The processing of the vertices consists in calculating a new label value corresponding to the most present label in the neighborhood of the vertex. In case of a tie, a random choice is made. Even though the visit order is based on the criticality of the vertices, the groups computed via the label propagation algorithm can reach the capacity constraint while cutting a critical path. The label propagation algorithm favors grouping communities, whereas, in our case, the objective is to group vertices along critical paths.

Another approach is to apply acyclic partitioning to a DAG or DAH. Acyclic partitioning was first defined by J. Cong *et al.* [44]. J. Hermann *et al.* [91, 92] presented an algorithm for acyclic graph partitioning, and M. Popp *et al.* [153] extended it to hypergraph partitioning. J. Nossack *et al.* [140] introduced an exact algorithm based on a branch-and-bound method for the acyclic partitioning problem. Their aim was to minimize the number of parts traversed by a path in the DAG/DAH. In our context, non-critical paths can traverse the same part several times without affecting the value of the partitioned critical path, so it is not mandatory to apply this constraint to all paths.

The following works are available to readers willing to learn more about acyclic partitioning [44, 91, 92, 136, 140, 153, 189], and a recent survey on hypergraph partitioning can be found in [35].

5.2 Traversal algorithms

Traversal algorithms use vertex neighborhood to explore the hypergraph. Traversal algorithms have a linear complexity with respect to the number of vertices and hyperedges, hence in $O(|V| + |A|)$ time. For example, a breadth-first traversal algorithm groups vertices using their neighborhood, creating at least a contiguous partition for the first part. Depending on the topology of the red-black hypergraph, the critical path can be minimized by taking advantage of contiguous partitions. One way to improve the length of the path during partitioning is to modify the processing order of vertices, hence only the most critical vertices are explored, rather than a random neighbor. This allows all vertices of a critical path to be

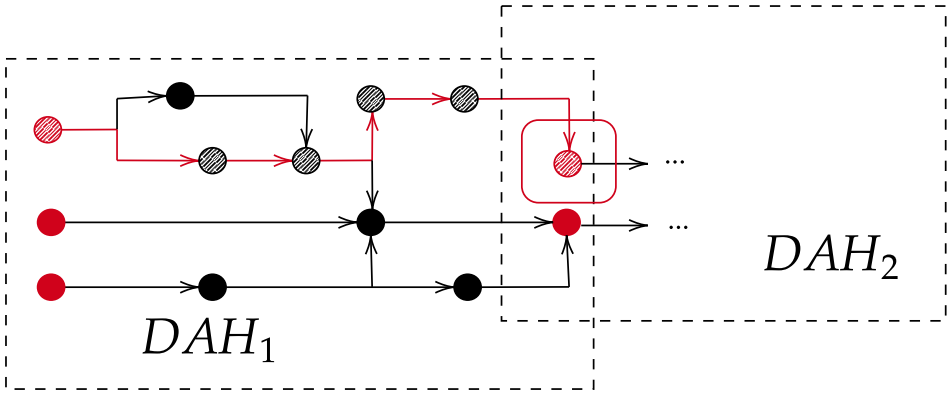


Figure 5.1: In this example, two connected DAHs are represented. DAH_1 and DAH_2 are connected since they share red vertices. Each striped vertex is critical. Specifically, the red vertex in DAH_1 and DAH_2 is critical in both DAHs. To avoid cutting this vertex, the exploration should continue in DAH_2 even if all vertices in DAH_1 have not all been visited. Finally, this example highlights the use of exploration through each DAH when a critical path shares a red vertex in two DAHs.

grouped in the same part, if the capacity constraint allows for it. More details on this approach are given in Subsection 5.2.1. Critical paths sometimes share a red vertex, which constitutes a sink in one DAH and a source in another DAH, as shown in Figure 5.1. In this case, continuing exploration beyond the currently explored DAH is advantageous. For this reason, we have investigated an approach using a dedicated depth-first search algorithm. More details on this approach are provided in Subsection 5.2.2.

5.2.1 Initial partitioning based on breadth-first search driven by vertex criticality

The breadth-first search algorithm was first introduced in 1945 in the rejected doctoral thesis of Konrad Zuse. This algorithm, which was designed for the search for connected components in graphs, was finally published in 1972 [195]. However, Edward F. Moore [135] had rediscovered the algorithm in 1959, for finding the shortest path in a maze. C. Y. Lee [123] also developed it in 1961, in a wire routing algorithm for electronic circuits.

Contiguous partitioning has been studied for graphs by B. W. Kernighan [113], and extended to hypergraphs by A. Grandjean *et al.* [86]. One instance of this problem is program partitioning. In the problem defined by B. W. Kernighan [113], a program is modeled as a graph in which the vertices represent instructions and the edges represent possible successors. Each edge weight models the relative frequencies of transitions. Hence, partitioning the instruction graph is equivalent of dividing the program into pages of fixed size while minimizing the frequency of interpage transitions. A. Grandjean *et al.* [86] presented the problem in matrix partitioning such that if columns i and j are in a same part k, all columns between i and j are in part k.

Contiguity along critical paths in a DAH is an interesting approach for red-black hypergraph partitioning, because it limits the cuts along critical paths.

Following these lines of thought, we designed an algorithm based on a breadth-first search algorithm and driven by the criticality of the vertices. The formulas for calculating the criticality of the vertices have been defined in Subsection 4.2.2. The strategy proposed here is to use vertex criticality to prioritize visits to critical neighbors, in contrast to traversal algorithms that do not prioritize neighbors. This strategy makes it possible to go deeper into the DAH, to group vertices along a locally critical path. Grouping by vertex criticality increases the likelihood that the vertices along this path will be placed in the same part.

Algorithm 6, which we called the Derived Breadth-First-Search (DBFS) algorithm, is an initial partitioning algorithm which is based on a search along the vertices of H, driven by criticality. It considers neighbor vertices of each vertex v according to their criticality value $r(v)$, so as to avoid multiple cuts along critical paths. In our context, the multiple cuts constraint is also important, because we must avoid cutting the same path multiple times. Indeed, the cut cost D_{ij} is often larger than the path length. Our algorithm considers the criticality of each vertex v, and of its neighbors. Each vertex is inserted in a priority queue ordered by vertex criticality. To select vertices to consider, we use the value of their in-degrees $\delta^-(v)$, which is decremented for all outgoing neighbors of v when v is visited. This allows one to obtain a topology-driven hypergraph traversal, considering the criticality of the vertices. The exploration starts from the red vertices which are the sources of the DAH. The algorithm explores the vertices in each DAH and inserts the visited vertices into an array that records the current state of the partition. To explore each DAH, our algorithm manages two priority queues: `p_queue` and `p_queue′`. The queue `p_queue′` processes the source and black vertices of the DAH to be explored, and `p_queue`, considered as the main queue, processes the red vertices that are the sinks of the explored DAH and sources of subsequent DAHs to explore. This main queue ensures that the algorithm does not start exploring another DAH while there are still vertices to visit in the current DAH. The two

priority queues store vertices according to their criticality value, by inserting the most critical vertices at first. When a vertex v is processed, $r(v)$ is the maximum criticality of all the queues' vertices. The queues are empty at the beginning of the algorithm, and every insertion is performed in $O(log_2(|V|))$ time by using a heap data structure.

Lemma 5.2.1. *Let $p \in P^R$ be a red-red path, and a path order $<_p$. For all $u, v \in p$, if $u <_p v$, then $r(u) \geq r(v)$.*

Theorem 5.2.1. *Let $H = (V, A)$ be a DAH, and ov, a visit order computed by DBFS. According to the derived breadth-first-search, it is not possible to have a pattern v_a, v_b, v_c, with $v_a, v_b, v_c \in V^B$, such that $r(v_a) > r(v_b)$ and $r(v_c) \geq r(v_a)$.*

Proof. The traversal starts from at least one source of the DAH H, so $\exists v_R \in V^R$ such that $v_R <_{ov} v_a$. According to Lemma 5.2.1, we have $r(v_R) \geq r(v_a)$. Let us assume a pattern v_a, v_b, v_c, with $v_a, v_b, v_c \in V^B$, $r(v_a) > r(v_b)$ and $r(v_c) \geq r(v_a)$. As the vertices are black vertices of the same DAH, each of them will be inserted in the secondary priority queue according to their criticality value, which is a contradiction. □

Theorem 5.2.1 states that the DBFS algorithm allows one to perform a walk following the local topological order by selecting, at each step, a neighbor of maximum criticality. This choice allows one to favor the grouping, within the same part, of neighboring vertices with high criticality. As long as the size constraint is respected, every selected vertex will be placed in the same part. An example can be found in Figure 5.2.

Lemma 5.2.2. *The Derived Breadth-First Search Algorithm 6 runs in $O(|V|log_2(|V|) + \Lambda|A|)$ time, with $|V|$ the number of vertices, $|A|$ the number of hyperarcs, and Λ the maximum size of hyperarcs.*

Proof. The algorithm performs a breadth-first search, such that vertices are visited only once. The `is_visited` array ensures the following invariant: when some vertex is visited, it is marked with the value `True` and is no longer processed, as indicated by the condition at line 25. However, the algorithm has two `while` loops at lines 11 and 14, which depend on two queues. The array `is_visited` ensures that the vertices are only processed once, and the whole of these two loops is executed in $O(|V|)$ time. In the second `while` loop, there is a `for` loop at line 23. This `for` loop iterates over the hyperarcs of which v is a source. We assume that each hyperarc is visited only once because there is exactly one source per hyperarc. Every non-visited vertex in the current hyperarc is inserted in a priority queue encoded by a heap data structure. The time complexity for each insertion is in $O(\log_2(|V|))$. Each vertex is inserted and processed only once in the `while` loop

Algorithm 6 Initial partitioning algorithm based on Derived Breadth-First-Search driven by node criticality (DBFS)

Require: $H = (V, A, W_v, W_e)$ a red-black hypergraph, $r()$, vertex criticality function, k, the number of parts, M, the capacity constraint
Ensure: Π a k-partition of H

```
p_queue ← []  ▷ p_queue is a priority queue ordered by vertex criticality in decreasing order
is_visited ← []
for v ∈ V do
    if δ⁻(v) = 0 then
        insert_by_criticality(p_queue, v, r(v))  ▷ v is a source and v is inserted in p_queue according to its criticality r(v)
    end if
    is_visited[v] ← False  ▷ v is marked as visited to avoid visiting v twice
end for
size ← 0
i ← 0
while |p_queue| > 0 do
    p_queue′ ← p_queue
    p_queue ← []
    while |p_queue′| > 0 do  ▷ Exploring while a v ∈ V^B is queued. Red vertices are queued in p_queue
        v ← p_queue′.pop()  ▷ Pop the first vertex with highest criticality
        is_visited[v] ← True
        if size + W_v(v) > M then  ▷ When the constraint size is reached, the algorithm places the next vertices in the next empty part
            size ← 0
            i ← i + 1
        end if
        Π[v] ← i  ▷ v is in part i
        size ← size + W_v(v)  ▷ The weight of v is added to the weight of part i
        for a ∈ A(v) do  ▷ Visit the neighborhood of v
            for v′ ∈ a do
                if ¬is_visited[v′] then
                    is_visited[v′] ← True
```

27: **if** ¬is_red[v'] **then**
28: insert_by_criticality(p_queue$'$, v', $r(v')$) ▷ If v' is black, its Γ^+ neighbors are in the current DAH. Hence, v' is placed in the current queue to continue exploring the current DAH
29: **else**
30: insert_by_criticality(p_queue, v', $r(v')$) ▷ If v' is red, its Γ^+ neighbors are in different DAHs and will be explored next
31: **end if**
32: **end if**
33: **end for**
34: **end for**
35: **end while**
36: **end while**
37: **return** Π

at line 11. Moreover, to explore the neighborhood of each vertex, each hyperarc containing this vertex is visited. Let $\Lambda = \max\{|a|,\ \forall a \in A\}$ be the maximum size of hyperarcs; then, the total time complexity is $O(|V|\log_2(|V|) + \Lambda|A|)$, with the term $|V|\log_2(|V|)$ corresponding to the processing and insertion of vertices, and the term $\Lambda|A|$ corresponding to neighborhood exploration. Note that the Derived-Breadth-First algorithm has an additional factor of $\log_2(|V|)$ compared to the complexity of the Breadth-First search algorithm. This additional factor corresponds to the management of the priority queue in DBFS, which characterizes our algorithm, which aims to process critical vertices first.

□

5.2.2 Initial partitioning based on Depth-First Search driven by vertex criticality

C. P. Trémaux introduced one of the first versions of the depth-first search algorithm in the 19th century, as a strategy for solving mazes [71]. The maze is modeled as a graph, and the algorithm tries to find a sink, that is, the exit from the maze. The depth-first traversal algorithm processes vertices from neighbor to neighbor until it finds a sink or a previously visited vertex. If the sink is found, then the exit is found. Otherwise, the algorithm treats the unprocessed neighbors as a backward step in the maze. As pointed out in the previous chapters, digital electronic circuits are also objects that can be modeled by graphs and hypergraphs for partitioning. Therefore, the literature naturally contains works that take advantage of graph algorithms [30, 101], such as traversal algorithms. R. Burra *et al.* [30] presented a clustering approach based on a depth-first search, where ver-

tices are grouped according to their attractiveness. The attractiveness of vertices is defined by the Scaled Cost plus Min Perimeter (SCMP) ordering metric proposed by Kahng *et al.* [101]. In order to favor nodes with higher cumulative delay values, the authors proposed a combination of SCMP and delay objectives. As said in Kahng *et al.* [101]:

> "[...] Note also that our particular SCMP ordering is designed with respect to the stated problem formulation. For specific multi-FPGA system designs, other objectives may prevail [...]."

Indeed, we have chosen vertex criticality as our measure of attractiveness. Our primary goal is to minimize the degradation of critical paths when partitioning the associated red-black hypergraph.

For red-black hypergraphs composed of multiple DAHs, some paths may share a red vertex which constitutes the sink of one DAH and the source of another. When two critical red-red paths share a red vertex present in two or more DAHs, it may be more advantageous to group the vertices along these paths. Our Algorithm 7 takes advantage of the deep exploration of DAHs by vertex criticality, to group critical vertices within the same part.

The Derived Depth-First Search (DDFS) initial partitioning method, presented as Algorithm 7, performs a depth-first traversal driven by the criticality of the vertices. The main difference with the previous method is that all vertices will be inserted in the same priority queue, regardless of whether they are red or black. It enables a visit order concerning the criticality value, but does not consider the topological structure of the DAHs in H. The main idea behind this method is to be able to pack interconnected critical paths of several DAHs in the traversal order. The most critical neighboring path will be placed in the same part, as long as the capacity constraint is respected. However, DDFS may induce cuts within the DAHs, and yield a possible path cost degradation for paths of smaller criticality. The relative efficiency of DBFS and DDFS is likely to vary, depending on circuit topologies and on the distributions of path lengths.

A hint on the respective merits and drawbacks of DDFS and DBFS is presented in Figure 5.2.

Theorem 5.2.2. *Let p and p' be the connected longest paths of two DAHs H and H', with $p \cap p' = \{v_R\}$, $v_R \in V^R \cap V'^R$. Let v be the last black vertex along p and v' be the first black vertex along p', with $r(v) \leq r(v')$. Let ov be a visiting order computed by DDFS. According to the derived depth-first search, it is not possible to have a pattern $\{v, v_R, v'', \ldots, v'\}$ with $r(v'') > r(v')$ iff p' is a local maximal.*

Proof. $\Longrightarrow$ Suppose that there exists a path $\{v, v_R, v'', \ldots, v'\}$ with $r(v'') > r(v')$. Let p and p' be connected paths with $p \cap p' = \{v_R\}$, $v_R \in V^R \cap V'^R$ and p' being

Algorithm 7 Initial partitioning algorithm based on Derived Depth-First-Search driven by vertex criticality (DDFS)

Require: $H = (V, A, W_v, W_e)$ a red-black hypergraph, $r()$, the vertex criticality function, k, the number of parts, M, the capacity constraint
Ensure: Π a k-partition of H

```
p_queue ← []
is_visited ← []
for v ∈ V do
    if δ⁻(v) = 0 then
        insert_by_criticality(p_queue, v, r(v))        ▷ v is a source and v is inserted in p_queue according to its criticality r(v)
    end if
    is_visited[v] ← False      ▷ v is marked as visited to avoid visiting v twice
end for
size ← 0
i ← 0
while |p_queue| > 0 do
    v ← p_queue.pop()                                  ▷ Pop the first vertex in the queue
    is_visited[v] ← True
    if size + W_v(v) > M then      ▷ When the constraint size is reached, the algorithm places the next vertices in the next part
        size ← 0
        i ← i + 1
    end if
    Π[v] ← i                                           ▷ v is in part i
    size ← size + W_v(v)     ▷ The weight of v is added to the weight of part i
    for a ∈ A(v) do                                    ▷ Visit the neighborhood of v
        for v' ∈ a do
            if ¬is_visited[v'] then
                is_visited[v'] ← True
                insert_by_criticality(p_queue, v', r(v'))     ▷ If v' is not visited, v' is placed in the current queue to continue to explore H
            end if
        end for
    end for
end while
return Π
```

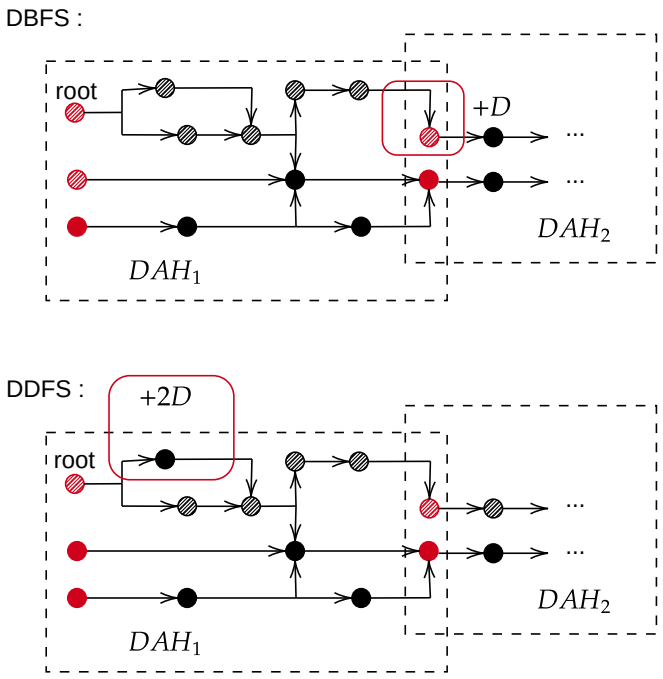


Figure 5.2: Examples of specific cut penalties for the DBFS and DDFS algorithms. Penalties are represented inside boxes. All hatched nodes will be placed in the same part. Each traversal starts from the "root" vertex. DBFS avoids multiple cuts along paths within a DAH, while DDFS does not (see the frame in the DDFS example). However, if a critical path starts in DAH_2 from a sink of DAH_1, DBFS can produce a cut along this critical path, while DDFS cannot (see the frame in the DBFS example).

a local critical path. A local critical path is a path with a maximal criticality value in a sub-hypergraph. According to Algorithm 7, each vertex of p' is inserted into the priority queue. However, the insertion is driven by vertex criticality. If $\exists v''$ such that $r(v'') > r(v')$ then, $\exists p'' \neq p'$ with $v'' \in p''$ such that p'' is a local maximum and p' is not a local maximum. That is a contradiction.

$\Longleftarrow$ Suppose that there exists a path p', which is a local critical path. As v' is the first black vertex in p', $r(v')$ is equal to the local maximal criticality. That is, $\nexists v''$ such that $r(v'') > r(v')$. That is a contradiction, because $r(v'') > r(v')$.

As the derived depth-first search visits the vertices with higher criticality first, v' will be visited after v_R. That is, a path $\{v, v_R, v'', \ldots, v'\}$ with $r(v'') > r(v')$ cannot exist.

□

Lemma 5.2.3. *The Derived Depth-First Search Algorithm 7 runs in $O(|V|log_2(|V|)+\Lambda|A|)$ time, with $|V|$ being the number of vertices, $|A|$ the number of hyperarcs, and Λ the maximum size of hyperarcs.*

Proof. The algorithm performs a depth-first search, hence vertices are visited only once. The `is_visited` array ensures the following invariant: if some vertex is visited, it is flagged as `True` and is no longer processed, as prescribed by the condition at line 22. However, the algorithm has a `while` loop at line 11, which depend on `p_queue`. The `while` loop contains a `for` loop at line 20. This `for` loop iterates over the hyperarcs of which v is a source. We assume that each hyperarc is visited only once, because there is exactly one source per hyperarc. Each non-visited vertex in the hyperarcs is inserted in a priority queue, encoded by a heap data structure. The complexity for each insertion is in $O(\log_2(|V|))$ time.

Each vertex is inserted and processed only once in the `while` loop at line 11. Moreover, to explore the neighborhood of each vertex, each hyperarc is visited. Let $\Lambda = \max\{|a|,\ \forall a \in A\}$ be the maximum size of hyperarcs; then, the total time complexity is $O(|V|\log_2(|V|) + \Lambda|A|)$, with $|V|\log_2(|V|)$ corresponding to the processing and insertion of vertices, and $\Lambda|A|$ corresponding to neighborhood exploration.

Note that the Derived-Depth-First algorithm has an additional factor of $\log_2(|V|)$ compared to the complexity of the Depth-First search algorithm. This additional factor corresponds to the management of the priority queue in DDFS, which characterizes our algorithm, which aims to process critical vertices first.

□

5.2.3 Critical Connected Components/Cone Partitioning

G. Saucier *et al.* [171, 172], and D. Brasen *et al.* [26, 27] introduced *cone partitioning* for circuit partitioning. Circuits are modeled by DAGs, whose sources and sinks correspond to the inputs/outputs of the circuit, *i.e.*, red vertices. Note that a DAG corresponds to one combinatorial component, *i.e.*, a DAH in that red-black hypergraph. In order to handle circuits composed of multiple connected components, the algorithms must be adapted.

A *cone* defines a connected component made up of vertices that are reached by traversing the vertices of the hypergraph from a sink to the sources in the reversed hypergraph, as shown in Figure 5.3. For hypergraphs, a reversed hyperarcs is one in which the source becomes the sink and the sinks become the sources. Note that all paths between the sources and a given sink are contained within the cone formed by the sink. Therefore, placing a cone in a part ensures that the paths in the cone are not cut. However, there may be paths between multiple cones that are not in the same parts, hence these paths have been cut. A cone is used to capture

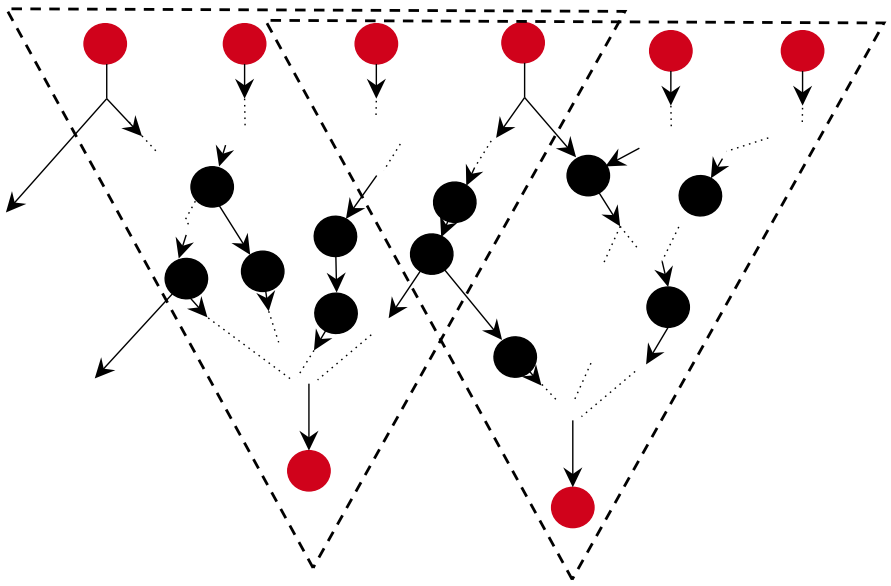


Figure 5.3: Example of a sub-DAH in which the sources are at the top, and the sinks, at the bottom. This example displays two cone components. A cone consists of a sink and all the vertices that can be reached from that sink in the reversed hypergraph, *i.e.*, the hypergraph whose arcs are reversed. In other words, a cone contains all paths that end in the sink of the cone. Note that a vertex which is not a sink can appear in several cones, as it is the case for the vertices in the middle of this figure.

a connected subset of the circuit. Because of these properties, cone partitioning seems worth exploiting. In addition, the number of cones may exceed the number of parts, because the number of cones depends on the number of sinks, whereas the number of parts is fixed. A merging process of cones must therefore by applied to obtain the desired number of parts.

In their work, G. Saucier *et al.* [171] present several algorithms for circuit partitioning based on cone structure. Each of these algorithms starts by calculating the cones in the DAG. Each component (cone) is evaluated as a function of the ratio between the number of vertices in the component, divided by the number of arcs between the component and the rest of the graph, *i.e.*, the size of the cocycle between the component and the rest of the graph. Let C_i and C_j be two cone components; the associated cost $cost(C_i, C_j)$, defined in [171], is:

$$cost(C_i, C_j) = \frac{|C_i \setminus V^R| + |C_j \setminus V^R|}{|\omega(C_i)| + |\omega(C_j)| - 2 \times |\omega(C_i) \cap \omega(C_j)| + 1} \ . \tag{5.1}$$

Note that in the original formula in [171], the denominator is set to: $|\omega(C_i)| + |\omega(C_j)| - 2 \times |\omega(C_i) \cap \omega(C_j)|$. Hence, if there are only two sinks, *i.e.*, two cones, in the graph, the formula cannot be computed because the denominator would be equal to 0. It is for this reason that we changed the denominator to $|\omega(C_i)| + |\omega(C_j)| - 2 \times |\omega(C_i) \cap \omega(C_j)| + 1$. This cost is used in [171] to measure the attraction between two components during a cone merging process. Indeed, as we have shown before, cones can share arcs. Hence, unmerged cones which share arcs result in cut arcs and thus cut paths. The selection of the cones to merge must therefore be optimized to satisfy the partitioning objective function. The authors also suggest grouping components that share a critical path if the capacity constraint allows for it. If a cone component is too large, the authors suggest partitioning the cone using the minimum cut metric.

Since, in this thesis, we are interested in the objective function f_p, we propose to extend the definition of cone connected component to that of critical connected components, according to the vertex criticality value $r(v)$ defined in Section 1.2. A *critical cone* is a cone whose size is fixed and whose vertices maximize criticality. Hence, let c be a cone and c' a critical cone, both originating from the same sink v. We have the following properties:

$$\begin{cases} c' \subset c \ , \\ \forall v \in c, \ \forall v' \in c' \ , \\ r(v) \leq r(v') \ . \end{cases}$$

These properties ensure that, when a critical cone is computed, the critical cone contains the vertices of the cone of highest criticality, thus maximizing the grouping of critical paths.

For this purpose, we propose the Connected Component Partitioning (CCP) Algorithm 8, in which we extend the computation of cone components according to the criticality of the vertices and a constraint size. As cone components can be large and group non-critical vertices together with critical vertices, the capacity limit is quickly reached, and it is no longer possible to continue grouping vertices. To tackle this problem, in Algorithm 8, we first process only the vertices with a criticality ranging between $\max_{v \in V} r(v) - bD$ and $\max_{v \in V} r(v)$, where D is the cost of a cut and b is an integer equal to one by default. This restriction, implemented in the condition at line 13, allows critical-related components to be calculated first. To compute the components, we use the union-find structure defined by R. E. Tarjan [186]. We then run the procedure a second time at line 26, without filtering the vertices.

The final number of components may not match the expected number of parts, because each cone corresponds to one sink. Consequently, if the number of sinks is greater than the number of parts, the number of components will be greater than the number of parts. To tackle this case, we defined a new reduced hypergraph such that each vertex of this new hypergraph represents a cone in the original hypergraph. Then, we partition this hypergraph. During the refinement phase, the partition is adjusted by swapping vertices across parts.

Lemma 5.2.4. *The Critical Connected Component Partitioning Algorithm 8 runs in $O(|V|^2)$, with $|V|$ the number of vertices, $|A|$ the number of hyperarcs, and Λ the maximum size of hyperarcs.*

Proof. For this complexity analysis, we will assume that `bound` is equal to 0. The algorithm performs a critical connected component search, so that vertices are visited only once. This process is repeated twice. The first pass considers vertices with a criticality greater than the value of the variable `bound`. The `is_visited` array ensures the following invariant: if the vertex is visited, it is flagged and is no longer processed, as prescribed by the condition at line 14. The algorithm has a `while` loop at line 10, which iterates over the `queue`. The `while` loop contains a `for` loop at line 12. This `for` loop iterates over the in-neighbors of the vertex being processed, and executes the `union` between two components, in $O(|V|)$ time. As $\sum_{v \in V} |\Gamma^-(v)|$ is in $O(V)$, the time complexity of the `for` loop is in $O(|V|^2)$. Therefore, the `while` loop operates in $O(|V|^3)$ time.

The `for` loop at line 4 runs in $O(|V^R|)$ time. Without loss of generality, we agree that the partitioning algorithm of the reduced red-black hypergraph is executed in at most $O(|V|)$ time. This gives a total complexity of $O(|V| + |V|^3)$ for the Critical Connected Component Partitioning Algorithm 8. Note that in the case of amortized complexity, the union operation at line 19 is performed in $O(\alpha(|V|))$ time [186], where α is the functional inverse of Ackermann's function and is very slow-growing. In practice, the integer $\alpha(|V|)$ is less than 5 for every value of

Algorithm 8 Initial partitioning algorithm based on Critical Connected Components/Cone Partitioning driven by vertex criticality

Require: $H = (V, A, W_v, W_e)$ a red-black hypergraph, k, the number of parts, bound, the cluster bound, ϵ, the balanced factor, M, the capacity constraint
Ensure: Π a k-partition of H

1: $C_v \leftarrow [v],\ \forall v \in V$ ▷ Union-find set structure
2: queue $\leftarrow$ []
3: is_visited $\leftarrow$ []
4: **for** $v_R \in V^R$ **do**
5: **if** $r(v_R) >$ bound $\wedge\ |\Gamma^-(v_R)| > 0$ **then** ▷ Each critical sink is queued
6: queue $\leftarrow$ queue $+ v$
7: **end if**
8: is_visited$[v] \leftarrow$ False
9: **end for**
10: **while** |queue| > 0 **do**
11: $v \leftarrow$ p_queue.pop() ▷ Pop the first vertex in the queue
12: **for** $v' \in \Gamma^-(v)$ **do** ▷ Incoming neighbors are processed to compute the cone
13: **if** $r(v') \geq$ bound **then** ▷ Only critical vertices are processed unless the bound is equal to 0
14: **if** $\neg$is_visited$[v']$ **then** ▷ If v' is not visited, v' is queued to visit its incoming neighbors in the next steps
15: is_visited$[v'] \leftarrow$ True
16: queue $\leftarrow$ queue $+ [v']$
17: **end if**
18: **if** size(C_v) + size$(C_{v'}) < M$ **then**
19: union$(C_v, C_{v'})$ ▷ If the constraint size is respected, we can merge C_v with $C_{v'}$
20: **end if**
21: **end if**
22: **end for**
23: **end while**
24: **if** bound > 0 **then**
25: bound $\leftarrow 0$ ▷ For the second pass, we set the bound to 0 to process all not yet visited vertices
26: "**go to**" line 3
27: **end if**
28: $\Pi \leftarrow$ partition(C)
29: **return** Π

$|V|$ [48]. We can therefore assume that the amortized complexity of the union operation is performed in constant time. Considering the complexity of the union, the final time complexity of the algorithm is in $O(|V|+\alpha(|V|)|V|^2) = O(|V|^2)$. □

5.2.4 Conclusion

In this Section, traversal algorithms have been adapted to the problem of partitioning with path cost minimization. Each of these algorithms has its own characteristics. DBFS explores the red-black hypergraph across DAHs, which is interesting for circuits made of large, highly connected DAHs. However, for very tightly connected circuits, DBFS is less interesting than DDFS, which explores the hypergraph in depth. In addition, DDFS traverses DAHs, making it effective when the critical paths of multiple DAHs share a red vertex.

Finally, we propose an extension of the cone partitioning algorithm that takes advantage of the criticality functions defined in the previous chapter. The structure of cones provides path grouping properties that are of interest for optimizing the f_p function. In addition, this algorithm locally groups critical components, making it more adaptable to different hypergraph than the more specific DBFS and DDFS algorithms. However, CCP does not produce the exact expected number of parts. To obtain the expected number of parts, CCP must be coupled with a partitioning algorithm.

5.3 Integer programming

Integer programming is often used to model problems before using exact integer program solvers. Several authors have used integer programming to tackle the problem of partitioning graphs [25, 76, 89, 139, 191], hypergraphs [119] and circuits [179]. Digital electronic circuits are usually of very large sizes, which means that partitions cannot be computed directly by way of exact solvers. However, we can reduce the size of the hypergraph, as in the multilevel scheme, to compute subsequently an initial partitioning of the smaller problem using an exact solver.

In this section, we introduce an integer programming approach to the red-black hypergraph partitioning problem. This model considers the minimization of both the cut and the degradation of critical path length during the partitioning step.

5.3.1 Model

The objective of the integer programming model is to minimize the degradation of the critical path. Therefore, one needs to compute the maximum degradation among all possible degradations. One also needs to model the target topology, to

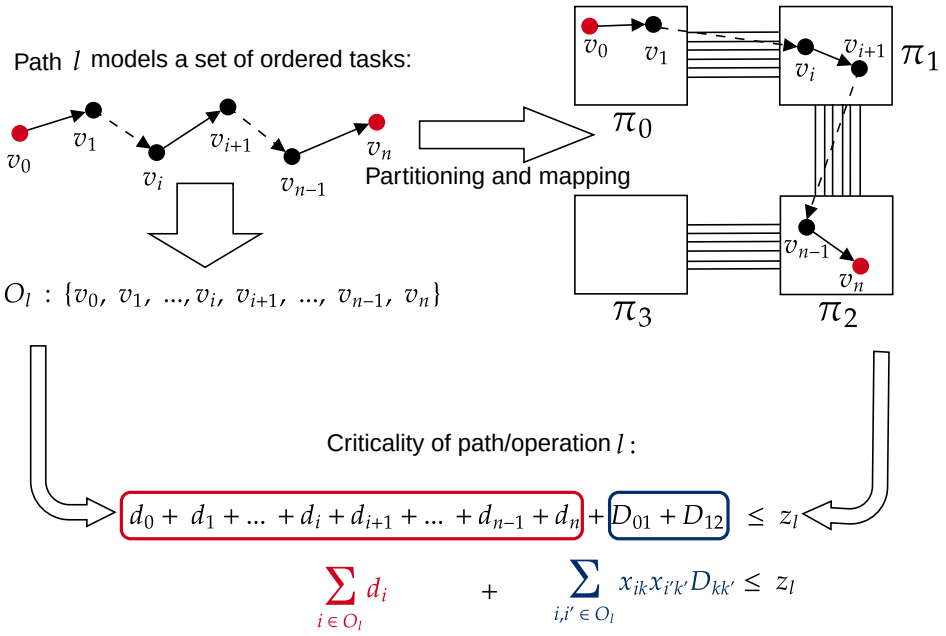


Figure 5.4: In this example, the red-red path l in a red-black hypergraph is defined as an ordered sequence of vertices. A set of tasks O_l is created from the path l. The scheduling constraint models the total completion time z_l of the task set O_l, *i.e.*, $\sum_{i \in O_l} d_i$, with an additional routing cost. This routing cost models the impact of the partitioning/mapping solution on the completion time of path l.

consider the potentially different delays between parts. Existing cut minimization tools do not address these two aspects of path length and target topology, since these tools only aim at reducing the connections between parts. As this objective is still essential in practice, we add a secondary objective to our model: minimizing the connectivity minus one.

Also, since the paths between two red vertices do not contain any cycle, it is possible to see the chain of black vertices in a path as a sequence of operations/tasks i associated with some job l. Consequently, we consider scheduling constraints in our model to minimize the impact of partitioning on the critical path. Given a path (job) $p = \{v_0, v_1, v_2\}$, the critical time associated with the path equals $\sum_{v \in p} d_v$. If vertices (tasks) belonging to p are placed in different parts, then a time penalty must be added to the total time of p. An example can be found in Figure 5.4.

Specifying formally our problem requires a lot of definitions, which are given in the following tables: sets can be found in Table 5.1, parameters in Table 5.2, and variables in Table 5.3.

Set	Definition
$\mathbf{V}$	Set of vertices.
$\mathbf{E}$	Set of hyperedges.
J	Set of jobs.
O_l	Ordered set of operations of job l, $(i \in O_l)$, where O_{l1} and O_{ln} are the first and the last elements of O_l.
i,i'	Vertices/operation index (i, $i' \in \mathbf{V}$).
j, j'	Hyperedges index ($j, j' \in \mathbf{E}$).
l	Job index ($l \in J$).
k	Part index.

Table 5.1: Definitions of indices and sets for specifying our integer programming problem.

Parameter	Definition
n	Number of vertices.
m	Number of hyperedges.
h_{ij}	1 if vertex i is connected to hyperedge j, 0 otherwise.
c_{kr}	Capacity of part k for resource r.
q_{ir}	Quantity of resource r, required by i.
d_i	Propagation time of vertices (operation) i.
$D_{k,k'}$	Delay between part k and k'.
$\mathcal{W}_v$	Vertex weight.
$\mathcal{W}_a$	Hyperedge weight.

Table 5.2: Definitions of parameters for specifying our integer programming problem.

The specification of our integer program, which is presented below, aims at fulfilling two objectives: 5.2a for critical path minimization, and 5.2b for connectivity

Variable	Definition
x_{ik}	1 iff vertex i is mapped onto part k, 0 otherwise.
y_{jk}	1 iff hyperedge j has a vertex placed on part k.
z_l	Completion time of job l.
$z_{\max}$	Maximum completion time of jobs.

Table 5.3: Definitions of variables for specifying our integer programming problem.

cost minimization:

$$\min z_{\max} \ , \tag{5.2a}$$

$$\min \sum_j \mathcal{W}_a^j \left(\sum_k y_{jk} - 1 \right) \ , \tag{5.2b}$$

$$\text{subject to :} \tag{5.2c}$$

$$\sum_k x_{ik} = 1, \quad \forall i \ , \tag{5.2d}$$

$$h_{ij} x_{ik} \leq y_{jk}, \quad \forall i, j, k \ , \tag{5.2e}$$

$$\sum_i q_{ir} x_{ik} \leq c_{kr}, \quad \forall k, r \ , \tag{5.2f}$$

$$\sum_{i \in O_l} d_i + \sum_{i,i' \in O_l} x_{ik} x_{i'k'} D_{kk'} \leq z_l, \quad \forall k, k', l \ , \tag{5.2g}$$

$$z_l \leq z_{\max}, \quad \forall l \ , \tag{5.2h}$$

$$x_{ik}, y_{jk} \in \{0, 1\}, z_l \in \mathbb{N} \quad \forall i, j, k, l \ . \tag{5.2i}$$

$$\tag{5.2j}$$

Constraint 5.2d states that each vertex is mapped onto one part. Constraint 5.2e guarantees that y_{jk} equals the connectivity cost associated with hyperedge j. Constraint 5.2f ensures the capacity constraint is respected. Constraints 5.2g and 5.2h determine the value of the delay of the job (path) and the maximum delay (*critical path*). Constraint 5.2i enforces the non-negativity and integrity conditions on the variable.

5.3.2 Symmetries

There are symmetries in the solution space of hypergraph partitioning for cut size minimization. Indeed, in the plain partitioning case, if ω hyperedges span across parts, ω remains unchanged regardless of the labels of the parts. However, in our

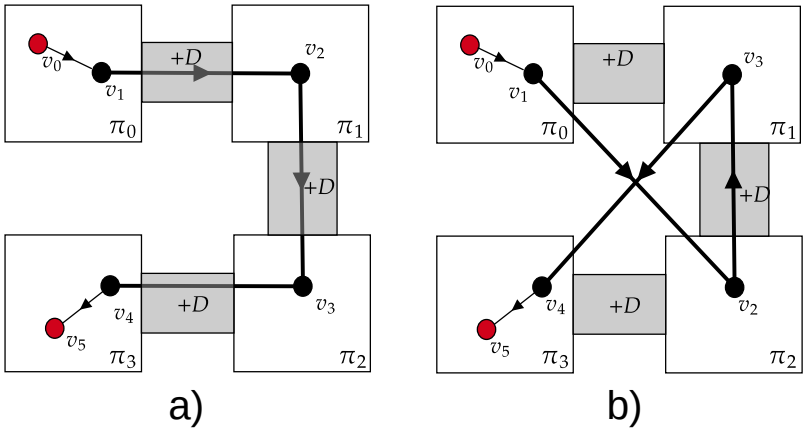


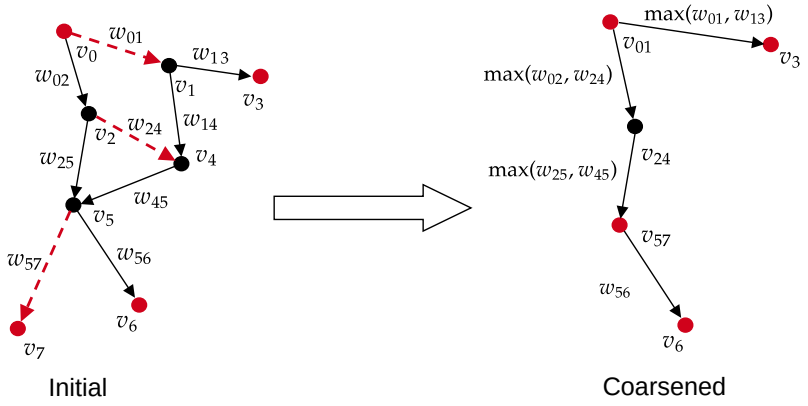
Figure 5.5: Path $p = \{v_0, v_1, v_2, v_3, v_4, v_5\}$ is mapped across 4 parts. For partition a, the path admits a routing penalty of $3D$, where D is the traversal time across parts. For partition b, the routing penalty is $5D$. Since there is no route between π_0 and π_2, we must necessarily pass through π_1, which yields a cost of $2D$ to go from π_0 to π_2. The same holds from π_1 to π_3. From the point of view of the size of the cut, partition a brings a cost of 3 cut edges, as does partition b. If we only consider the cut minimization objective, partitions a and b are symmetric.

problem, we are trying to minimize the path cost, which is degraded by routing paths across parts that are not always fully connected. An important point in modeling the partitioning problem is to break the symmetries. For example, P. Bonami *et al.* [25] propose models for the partitioning problem without symmetries. However, these constraints are too restrictive for the solution space associated with path cost, because, in our problem, the penalty between two parts is not homogeneous. Indeed, the target topology defines a time penalty associated with path routing. From a routing point of view, we cannot consider all partitions with the same subset of vertices but different labels, as a symmetry, an example is provided in Figure 5.5. Note that some symmetries still exist. For example, if we take the partition a shown in Figure 5.5, it is possible to create a partition a' by swapping the vertices of π_0 and π_3 and those of π_1 and π_2.

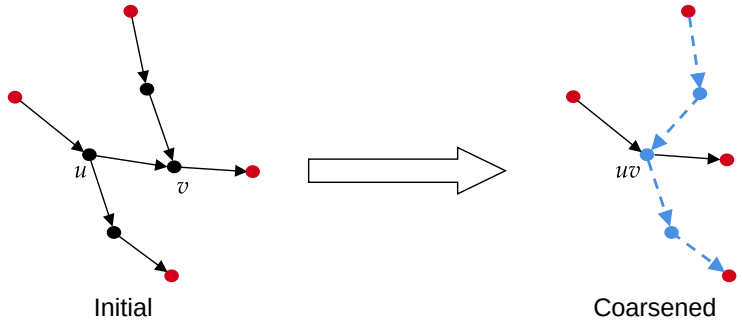
5.3.3 Path degradation

Using an exact solver is relevant only when the size of the instance is small. In the VLSI context, circuit sizes can be small or reduced by way of clustering. This is why we took the time to study the integer programming approach as an initial partitioning step. Before this step, a coarsening method is used to reduce

the instance size. During the coarsening phase, the structure of the red-black hypergraph can change, *i.e.*, extra connections between vertices can be created. As a result, the coarsened red-black hypergraph may have a modified critical path that does not exist in the original red-black hypergraph.



a) The merging of v_5 and v_7 reduces the length of path $\{v_0, v_2, v_5, v_7\}$.



b) The merging of u and v creates an erroneous path.

Figure 5.6: Two examples of possible path degradation during coarsening. In example a), each arc models the path relationship between the vertices. Dashed arcs are then merged. After merging, new paths are created, and some paths are deleted. For example, all paths ending in v_6 now end in v_{57} because v_{57} becomes a red vertex and v_7 is a red vertex (sink). In example b), merging the vertices u and v creates the dashed path that did not exist in the original instance.

Figure 5.6, presents such an example in which the value of the critical path in the coarsened red-black hypergraph is at least equal to the critical path in the

original hypergraph. In other words, the criticality of the paths in the coarsened red-black hypergraph is an upper bound of that in the initial hypergraph.

However, the new paths can be extremely approximated and may disrupt the initial partitioning step because integer programming optimizes job sequences which are biased by coarsening. There are several ways to overcome this problem. The first one is to create a new data structure based on the routing table. This solution creates a model with a much larger memory footprint. Another solution is to calculate a degradation tolerance factor for paths during the clustering phase. The tolerance factor may be used to model the additional cost of creating new paths. It might be interesting to experiment and measure the effectiveness of these two strategies in combination with our integer program as the first partitioning step. Maintaining the correctness of the path information during the initial partitioning would allow one to take advantage of an exact solver of a linear programming model.

5.3.4 Conclusion

In this section, we introduced an integer programming model, integrated into an exact approach, for solving concurrently the two problems of cut and path length minimization. Since circuits can be large, a coarsening step must be performed upstream to reduce the number of variables in the model. However, we have shown that the coarsening step can create erroneous paths when the vertices are merged. We have proposed strategies that may overcome this issue, but path modeling remains complex. Because digital electronic circuits can be very large, path information can be very degraded by the coarsening process. As a result, the initial solution produced by the linear program is likely to be approximate, which some how contradicts the purpose of this tool.

5.4 Mapping the initial partition

This section presents some approaches to map initial red-black hypergraph partitions onto a non-uniform topology. During this thesis, we did not have time to explore algorithms for optimizing the placement of a hypergraph partition onto a target topology. However, we have had some thoughts on the subject, which we summarize in this section.

Since many initial partitioning algorithms do not consider the target topology, an additional placement step is required, as shown in Figure 5.7. In various approaches to circuit partitioning, the partition is placed onto the target topology to minimize the impact on the partitioned critical path. S. Liou *et al.* [127] presented an algorithm for optimizing partition placement by considering the delay

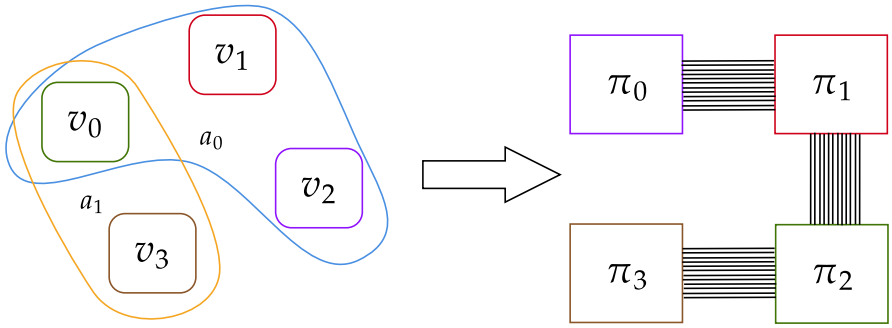


Figure 5.7: In this example, a 4-partition of a hypergraph is already computed. The original hypergraph is reduced to 4 vertices hypergraph H', in which each vertex is a part of the original hypergraph. At this step, the reduced hypergraph in the left is mapped onto the architecture in the right. Each vertex (part) is mapped onto a physical part π_i.

and signal capacities between FPGAs. Their algorithm swap parts over the FPGA platform to optimize placement.

It is possible to compute an "exact" solution for placing a partitioned hypergraph. First, the partitioned hypergraph is viewed as a graph in which each vertex correspond to a part. Then, for example, the integer programming model can be re-used again to move parts on the target topology, according to timing objectives. Finally, if the number of parts is too large, a heuristic based on the approach presented by S. Liou *et al.* or a refinement algorithm based on swapping of parts, may prove effective. The refinement algorithms then take over to optimize the placement at each level.

5.5 Experimental results

In this section we present some experimental results that we have achieved. These results have been made on the circuits and target topologies introduced in Chapter 3. Tables containing results presented in the following plots can been consulted in Appendix A.2.

In the experiments of this thesis, we decided to compare algorithms on metrics f_p and f_λ. The metric f_p calculates the critical path cost, which indicates at which frequency the partitioned circuit could operates. The second metric, f_λ, indicates

how many signals need to be transmitted across FPGAs. In practice, a large number of transmitted signals could implies signal multiplexing. Indeed, signal multiplexing is not a part of this thesis, but it is important to have a look on the effects of algorithms on f_λ metric.

To make our experiments on our initial partitioning algorithms DBFS, DDFS, and CCP, we implemented them in *C* programming language and compiled with `gcc` with flag `-O3`.

All our experiments were performed on one core of a machine that comprises 4 Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz, 16 GB of main memory, 8 Mib of L3-Cache, and 1 Mib of L2-Cache.

As the degradation between the parts can be non-homogeneous, we have defined several topologies composed of four and eight FPGAs. We also considered fully connected topologies, to highlight the advantage of our algorithms over standard partitioners. Obtaining the third-party tools to reproduce the results is difficult for most existing approaches. Since the existing approaches used partitions provided by cut minimization partitioning tools, we will do the same with our approaches considering the target topology.

5.5.1 Integer programming results

All of the results presented in this subsection have been published in our work [161]. To validate our models and algorithms, we have performed experiments on the ITC benchmark of logic circuits [50], detailed in Chapter 3. These circuits consist of acyclic combinatorial blocks, bounded by their input and output registers. Every combinatorial block can therefore be modeled as a DAH. Their computation time is conditioned by their critical path, defined as the longest path between two registers (*i.e.*, two red vertices). Our work aims at minimizing the degradation of the critical path during partitioning, with respect to the target topology. For instance, we arbitrarily set a unique traversal cost $d(v)$ of 0.58 ns for black vertices and 0.38 ns for red vertices, corresponding to typical traversal times for standard cells in common CMOS technologies. As the degradation between the parts can be non-homogeneous, we have defined a chain topology $\pi_0, \pi_1, \pi_2, \pi_3$ composed of four elements. To solve our integer programming problem, we use Gurobi Optimizer version 9.1.2 with a time limit set to $600s$. During the refinement phase, we use the DKFM algorithm. DKFM is a local search algorithm dedicated to minimizing path length presented in the next Chapter 6. We used khMetis, a k-way partitioner, rather than hMetis, because hMetis is based on recursive bipartitioning methods, which often do not respect the balance constraint. We used the maximum criticality scheme r^* as a weight for the hyperedges, to guide khMetis to minimize the number of cuts along the critical path as much as possible. The coarsening algorithm we used is HEM, introduced in Chapter 4.

Table 5.4: Results on path-cost (f_p) degradation factor of partitions compared to those of produced by khMetis.

Instance	khMetis	Multilevel+IP+DKFM
b01	1.0	**0.83**
b02	**1.0**	**1.0**
b03	1.0	**0.8**
b04	**1.0**	**1.0**
b05	**1.0**	**1.0**
b06	1.0	**0.75**
b07	1.0	**0.67**
b08	1.0	**0.77**
b09	**1.0**	**1.0**
b10	**1.0**	**1.0**
b11	1.0	**0.875**
b12	**1.0**	**1.0**
b13	1.0	**0.67**
b14	**1.0**	2.5
b17	**1.0**	1.08

Table 5.4 presents results of the degradation of f_p during partitioning in which our proposed methods seems to be better for a majority of circuits on path cost. Specifically, results shows that our approach gives better results for b01-13. Indeed, the first coarsening step allows the grouping of the most critical vertices, while maintaining a balance in the reduced hypergraph. Since the initial partitioning considers the topology, it allows for finding an appropriate placement before the refinement phase. For instances b14 and b17, the time limit set to 600s is possibly not sufficient for Gurobi to find a good solution. As we have shown previously, a method needs to be added during the coarsening step to better reduce the size of the instance while retaining sufficient criticality information for the integer program. Table 5.5 evidences that hMetis yields better minimum connectivity cost f_λ for all circuits except b03, b06 and b09. Note that our approach allows a better solution for both f_p and f_λ for three circuits, and we consider f_p as our first objective.

In this experiment, we measured the feasibility of initial partitioning by integer programming on a subset of circuits. Despite the path bias caused by the coarsening step, the results demonstrate how our methods minimize the f_p function well compared to the min-cut tool. However, as the hypergraph becomes larger,

Table 5.5: Results on connectivity-minus-one cost degradation factor of partitions compared to those of produced by khMetis.

Instance	khMetis	Multilevel+IP+DKFM
b01	**1.0**	1.55
b02	1.0	**0.78**
b03	**1.0**	1.40
b04	**1.0**	2.70
b05	**1.0**	2.28
b06	1.0	**0.92**
b07	**1.0**	3.76
b08	**1.0**	1.58
b09	1.0	**0.98**
b10	**1.0**	1.26
b11	**1.0**	1.14
b12	**1.0**	3.23
b13	**1.0**	1.31
b14	**1.0**	7.35
b17	**1.0**	3.72

the path information becomes less accurate. Therefore, for this method to be applicable to larger circuits, the coarsening step needs to be adjusted.

5.5.2 Results for DBFS, DDFS and CCP with min-cut tools

In this subsection we present results on circuits benchmarks ITC, Chipyard and Titan, targeting the 6 topologies described in Chapter 3. We compare both DBFS, DDFS and CCP with min-cut tools hMetis, khMetis, PaToH, KaHyPar, and TopoPart.

For our experimentations, we set the hMetis and khMetis balance parameter to 5% and the number of runs to 10. For coarsening, we chose the heavy edge strategies to take advantage of our weighting based on vertex criticality. We chose to use PaToH in its default version to see whether it could produce acceptable solutions in reasonable time, because computation time is a critical aspect of industrial-size circuit prototyping. We set the vertex visit order (VO) to *maximum net size sorted* mode and matching type (MT) to *heavy connectivity clustering*. This vertex visit order favors the processing of the largest hyperarcs first. In addition, we set the partitioning algorithm (PA) to *greedy hypergraph growing with*

max net. We set KKAHYPAR to use the connectivity-minus-one objective with the same parameters as indicated on the KAHYPAR webpage [1].

Each algorithm was run 10 times for each circuit, and we computed the relative degradation to evaluate the quality of a partition. Let H be an instance, $d_{\max}(H)$ its critical path and $d^{\Pi}_{\max}(H)$ the critical path of the partition Π. The relative degradation is calculated by the following formula:

$$\tau(H^{\Pi}) = \frac{(d^{\Pi}_{\max}(H) - d_{\max}(H))}{d_{\max}(H)} \quad . \tag{5.3}$$

Each figure presents critical path degradation relative to the best degradation produced. A $\tau(H^{\Pi})$ is equal to 0 when $d^{\Pi}_{\max}(H) = d_{\max}(H)$ and $\tau(H^{\Pi})$ is equal to 1 when $d^{\Pi}_{\max}(H) = 2 \times d_{\max}(H)$. Therefore, $\tau(H^{\Pi})+1$ is equal to the multiplicative coefficient of the value of the critical path $d_{\max}(H)$, that is:

$$d^{\Pi}_{\max}(H) = (\tau(H^{\Pi}) + 1) \times d_{\max}(H) \quad . \tag{5.4}$$

Each critical path degradation is sorted from the best to the worst, that is, each curve shows how much the algorithm is the best and how much it is the worst.

We decided to sort the relative degradation for each algorithm to show how much these algorithms can produce a small or high critical path degradation. In addition, the sorted degradation defines a curve that represents the algorithm's performance over the entire benchmark. For example, to determine whether an algorithm produces less degradation than the others, we need to look at the position of its plot in relation to the others. Thus, the further a plot associated with an algorithm is below the other plots, the better that algorithm performs. As a result, we can observe the increasing degradation over the whole benchmark for each target topology. Results on fully connected T3 and T6, K_4 and K_8, are presented in Figures 5.8 and 5.13. However, these plots cannot compare a result of an algorithm to a specific circuit to another one easily, but detailed results can are available in Appendix A.2.

In Figure 5.9 and Figure 5.10 allow one to compare critical path degradation between algorithms DBFS, DDFS and CCP with respect to min-cut petitioners, for a 4-partitioning. For these two topologies, DDFS produces fairly high degradation. However, if we look at Figure 5.8 presenting result for a fully connected target topology, DDFS produces better results, as at least it does not perform worse than KAHYPAR, PATOH, and DBFS. We can conclude that the poor results on topologies T1 and T2 are due to routing penalties induced by the notion of distance in the model.

Figure 5.8 allows us to estimate routing-related degradation for other topologies. Remember that none of these algorithms take the target topology into account. The results of Figure 5.8, show that CCP produces the least worst-case

degradation compared to the other algorithms while for both, topologies T1 and T2, not. We can conclude that CCP groups critical paths into cones and then merges them to limit the cut of critical or semi-critical paths. However, the degradation suffered on other topologies indicates that the resulting partition is less robust to routing than other partitions. This problem arises from the order in which the parts are assigned, from 0 to $k-1$. Indeed, this order does not depend on the target topology. Note that DBFS and DDFS have the same part assignment process.

DBFS and DDFS provide different results for a part of circuits; for example, for circuit B05, DBFS produces a degradation close to 1 while DDFS produces a degradation close to 3, but, for MNIST circuit, DDFS produces a degradation close to 4, which is better than the one produced by DBFS, close to 6. These differences highlight the importance of the choice between DBFS and DDFS for circuit partitioning. Note that for B05, CCP produces a degradation close to 2, which is between DBFS and DDFS. Hence, when circuit properties are unknown, we can compute partition from DBFS, DDFS and CCP, and select the best one.

Comparing the results obtained on Figure 5.13 against those of Figure 5.11 and Figure 5.12, one can see that CCP produces the lowest degradation. In the results regarding the T4 and T5 topologies, CCP produces the smallest degradation, but is not the best overall degradations. Whereas CCP is the best for a majority of circuit instances on the fully connected topology, with a maximum degradation of about 9 ($10 \times d_{\max}(H)$), one can see that PaToH produces also good results on topology T3 with a maximum degradation of about 9. In comparison, the next bests, khMetis and hMetis, yield a degradation close to 12 ($13 \times d_{\max}(H)$). It is worth noting that the greatest degradation occurred on the ITC circuits, which are smaller in size. In fact, since there are few paths, the critical path is very often cut, resulting in significant critical path degradation.

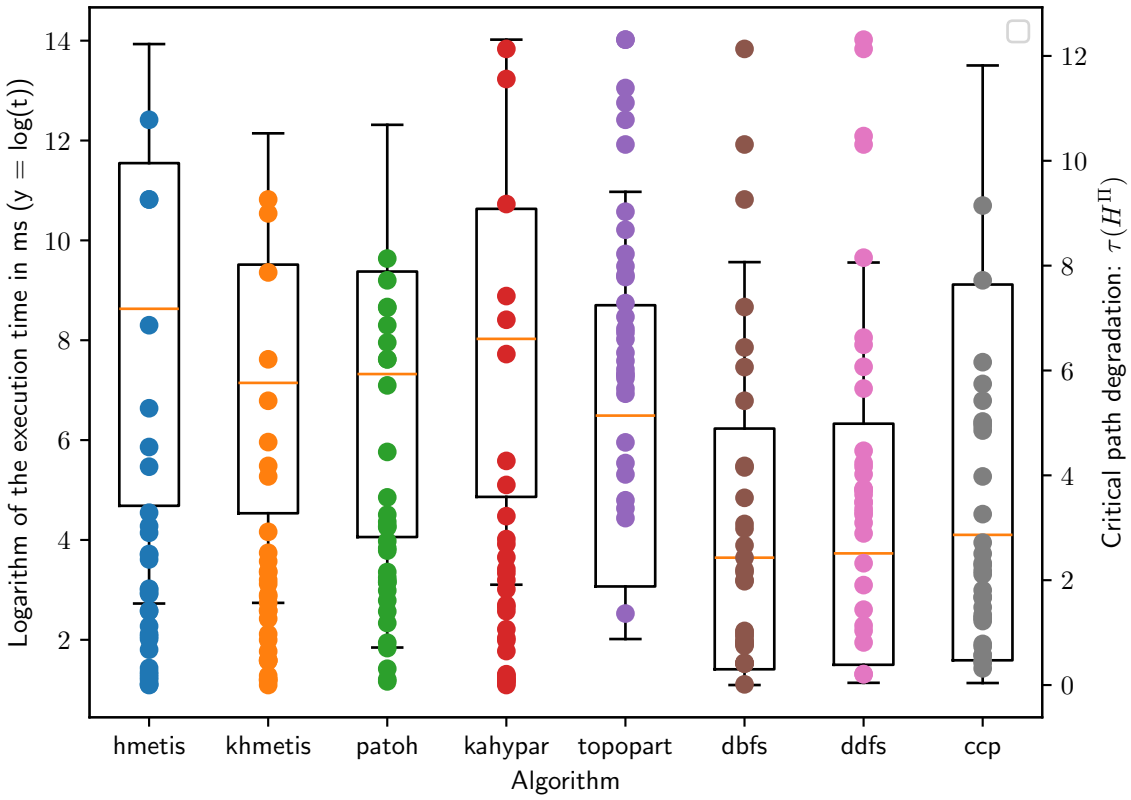


Figure 5.8: Results of degradation and execution time, produced by hMetis, khMetis, PaToH, KaHyPar, TopoPart, DBFS, DDFS and CCP when mapping onto the fully connected architecture T3. Each point is a circuit degradation corresponding to its algorithm. Each boxplot presents logarithm of the execution times in milliseconds of all circuits. For this topology, PaToH and CCP seems to perform better than others for critical path degradation. DBFS and DDFS are better for execution times. However, CCP have a good results on execution time too. Note that DBFS, DDFS, TopoPart and, KaHyPar have higher critical path degradation and KaHyPar, hMetis and khMetis have higher execution times.

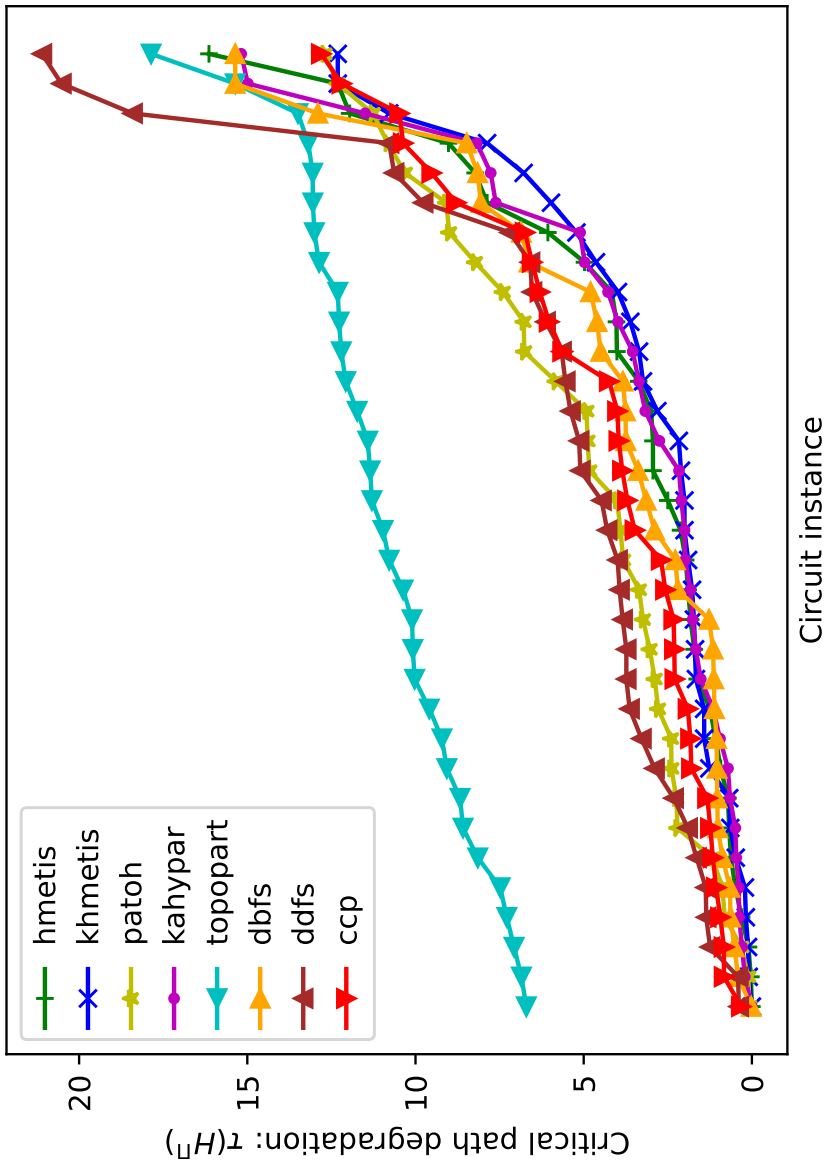


Figure 5.9: Results of degradation produced by hMetis, khMetis, PaToH, KaHyPar, DBFS, DDFS and CCP onto T1. For this topology, khMetis seems to perform better than others. DBFS is better for 13 circuits but its degradation reaches a value of 15 compared to khMetis and CCP, both close to 13. Note that DDFS end with higher degradation. However, TopoPart seems to have the worst performance on average.

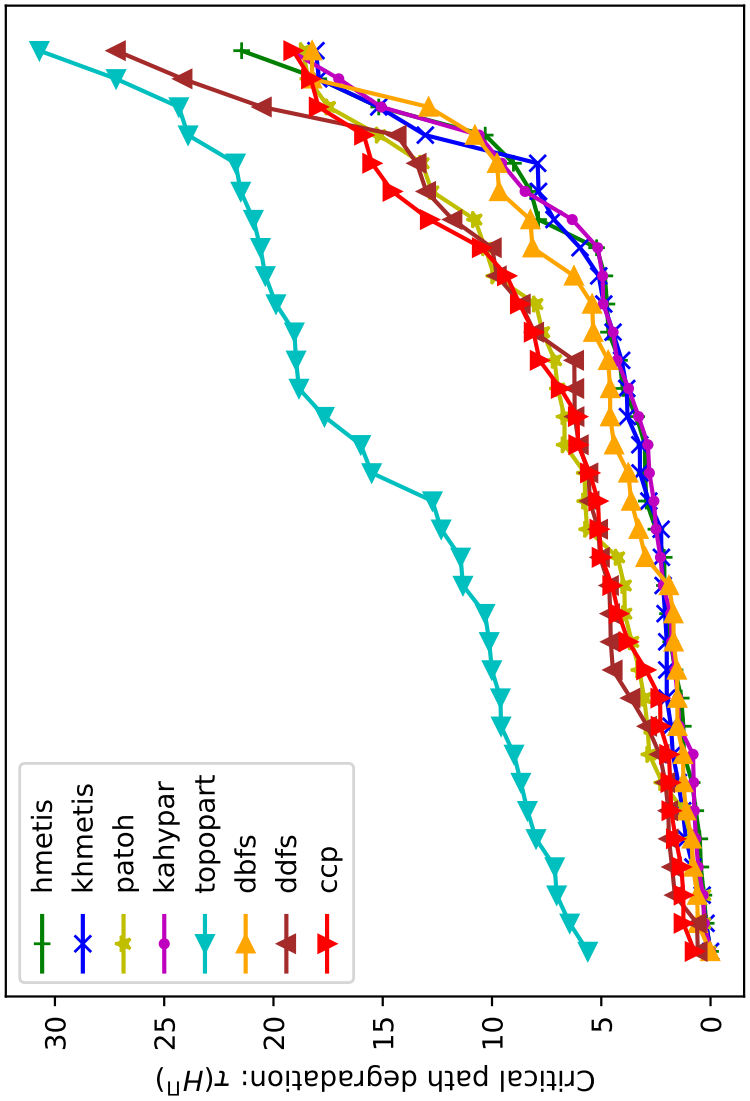


Figure 5.10: Results of degradation produced by hMetis, khMetis, PaToH, KaHyPar, TopoPart, DBFS, DDFS and CCP when mapping onto architecture T2. Note T2 is a form of a path with 4 FPGAs, hence, T2 causes more critical path degradation due to its topological structure. For this topology, hMetis, khMetis, KaHyPar and DBFS seem to perform better than the others. PaToH performs better for 3 circuits, but its degradation reaches a value of 5 for half of circuit benchmarks, compared to hMetis, khMetis, KaHyPar and DBFS, each below a value of 5 for 60% of circuits. Note that TopoPart end with higher degradation while all the others maximum degradation meet a value around of 18-19.

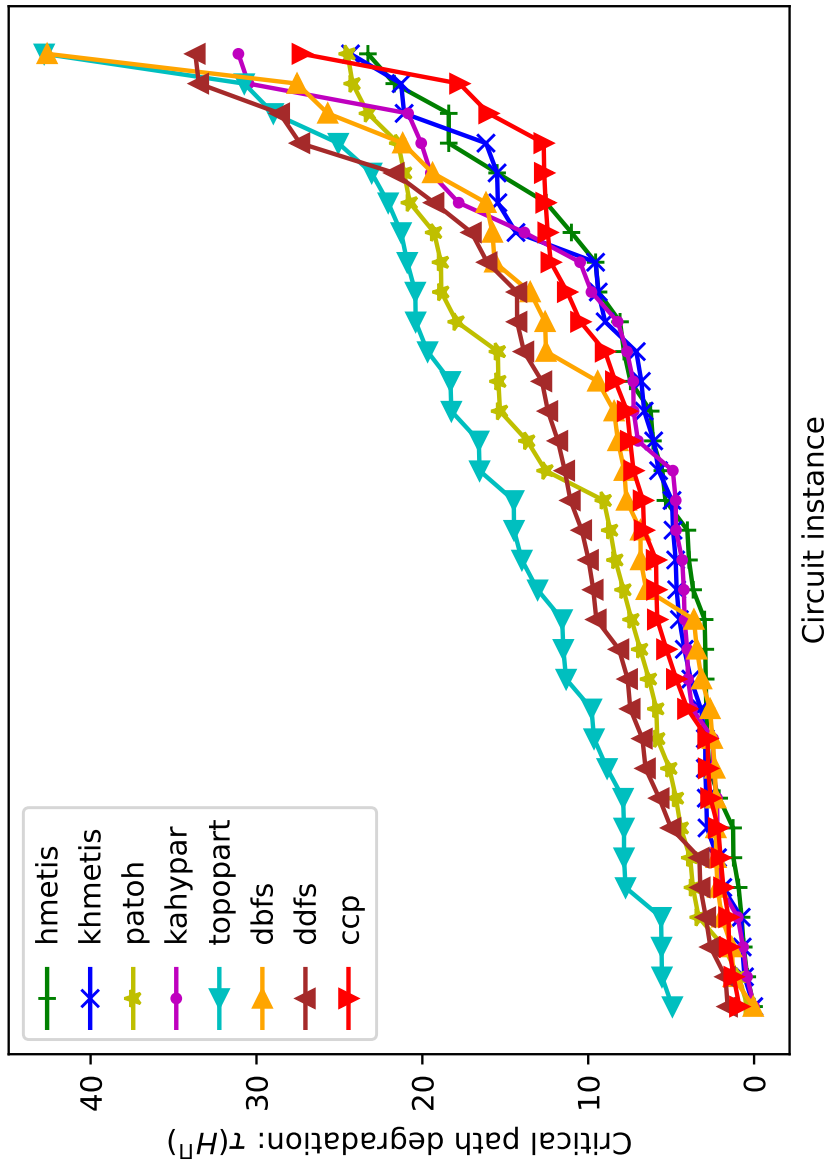


Figure 5.11: Results of degradation produced by hMetis, khMetis, PaToH, KaHyPar, DBFS, DDFS and CCP when mapping onto architecture T4. For this topology, hMetis, khMetis, and CCP seem to perform better than the others. PaToH reaches a degradation equal to a value of 5 for the first quartile of circuits compared to hMetis, khMetis, KaHyPar, each below a value of 5 for the quartile 2 (Q_2) of circuits. Note that DBFS and TopoPart ends with higher degradation up to a value of 43 while best maximum degradation meet around 25. This value results on multiple additional routing cost in small critical path, resulting in large degradation.

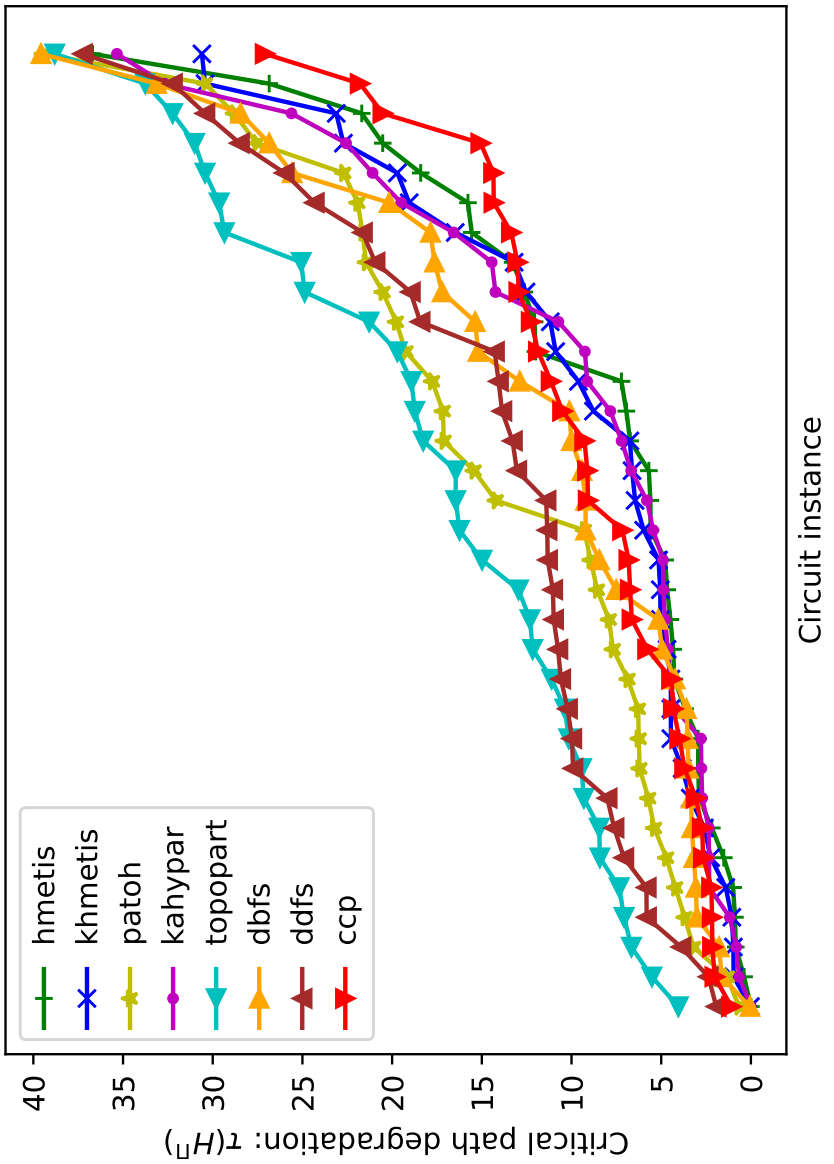


Figure 5.12: Results of degradation produced by hMetis, khMetis, PaToH, KaHyPar, DBFS, DDFS and CCP onto topology T5. Here, hMetis, and CCP seem to perform better than the others. Let us note that hMetis, khMetis, KaHyPar, DBFS and CCP, are each one below a value of 5 up to quartile 2 (Q_2) of circuits. Note that DBFS, TopoPart and PaToH end with higher degradation up to a value of 40 while best maximum degradation meet around 25.

The results for DBFS and DDFS presented in Figure 5.13 evidence the same behavior. It is important to note that DBFS performs better overall than DDFS. For example, B10 has a degradation of 6 with DBFS, versus 11 with DDFS; B12 has a degradation of 5 with DBFS, versus 8 with DDFS. Conversely, the MNIST circuit has a degradation of 5 with DDFS, versus 7 with DBFS. In line with what was presented in Section 5.2 on the differences between DBFS and DDFS, these examples show that, in practice, there are circuit examples where one of these two strategies is more advantageous than the others.

5.5.3 Results for connectivity cut cost

In this subsection, we focus on the connectivity-minus-one cost results. On multi-FPGA platforms, the number of wires in a connection between two FPGAs is limited. If the number of wires of circuit to be prototyped exceeds this limit, a multiplexing procedure is employed to use the interconnections multiple times. The effect of multiplexing is to add additional delay to the multiplexed paths. The subject of multiplexing is beyond the scope of this thesis and is usually dealt with in post-processing. However, we wanted to present results related to connectivity-minus-one cost, to show the possible discrepancy between min-cut algorithms and DBFS, DDFS, and CCP. To evaluate connectivity-minus-one cost of some partition Π, we compute a relative cost as follows:

$$\tau_\lambda(H^\Pi) = \frac{\omega(\Pi)}{\min\limits_{\text{algo}\in\text{ALGO}}\{\omega(\Pi^{\text{algo}})\}} \ , \tag{5.5}$$

with ALGO being the set of algorithms to compare. We chose to present this cost on a logarithmic scale because DBFS and DDFS typically produce partitions whose connectivity-minus-one can be one order of magnitude higher than the best cost, resulting in large relative costs. As a results, $\log(\tau_\lambda(H^\Pi)) = 0$ indicates a relative cut cost close to the best cut cost with $\tau_\lambda(H^\Pi) = 1$.

In this set of algorithms, only TopoPart computes a partition while taking into account the target topology. Algorithms that take the target topology into account will not produce the same partitions for each topology, unlike the others mincut tools. For this reason, we decided to evaluate the algorithms on topologies that are fully connected for connectivity-minus-one cost. Additional results on this metric on other target topologies can be found in the Appendix A.2.

Figure 5.14 and Figure 5.15, hMetis, KaHyPar, and khMetis produce partitions with best connectivity-minus-one compared to TopoPart, DBFS, DDFS and CCP. We note that PaToH yields good connectivity-minus-one costs for half of the circuit instances. Tables of numerical results can be consulted in Appendix A.2.

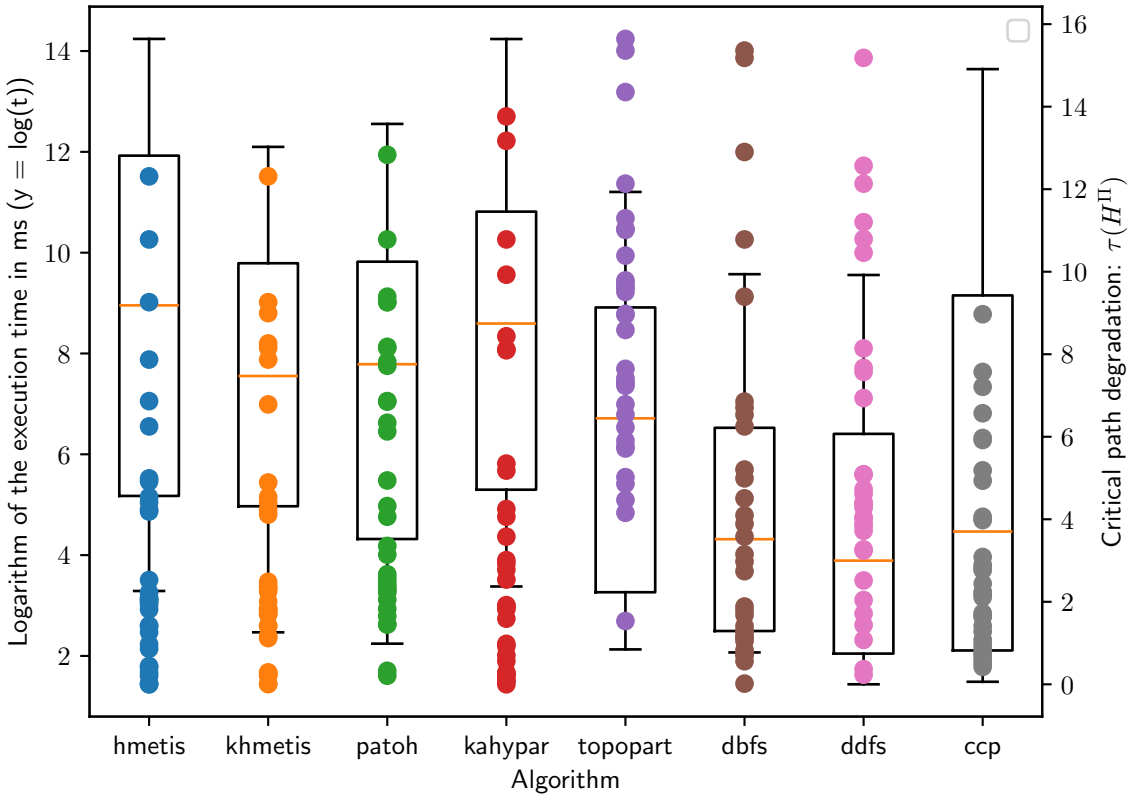


Figure 5.13: Results of degradation and execution time, produced by hMetis, khMetis, PaToH, KaHyPar, DBFS, DDFS and CCP onto the fully connected topology T6 composed of 8 FPGAs. Each point represents a circuit degradation for the corresponding algorithm. Each boxplot presents logarithm of the execution times in milliseconds for all circuits. For this topology, CCP seems to perform better than other algorithms with respect to critical path degradation, while DBFS and DDFS are faster. However, CCP is also quite fast as well. Note that DBFS, DDFS, PaToH and KaHyPar yield higher critical path degradation and KaHyPar, hMetis and are slower tha the others.

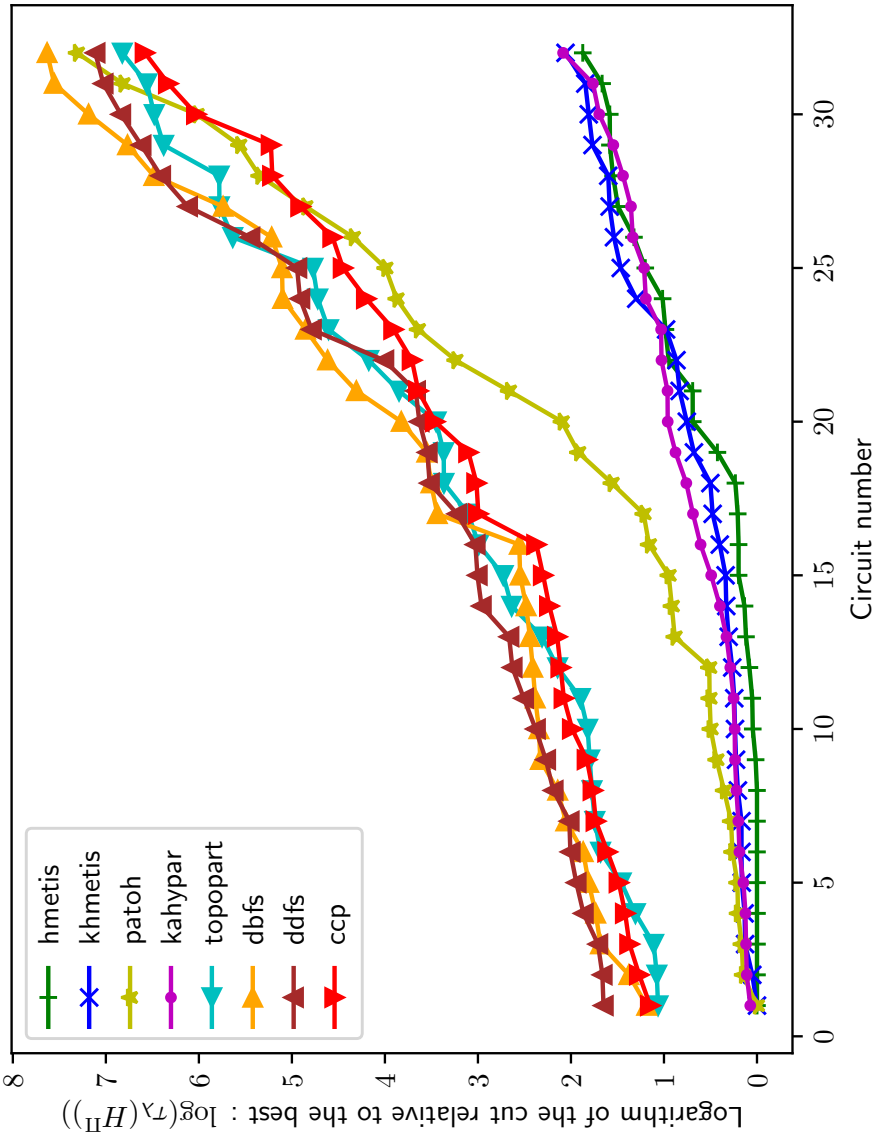


Figure 5.14: Logarithm of connectivity-minus-one relative to the best results when mapping onto architecture T3 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and greedy path cost algorithms: DBFS, DDFS and CCP. We notice a break between mincut and greedy algorithms. However, TopoPart behaves like DBFS, DDFS, and CCP, while PaToH produces good results for half of the circuits. DBFS produces worst connectivity-minus-one results.

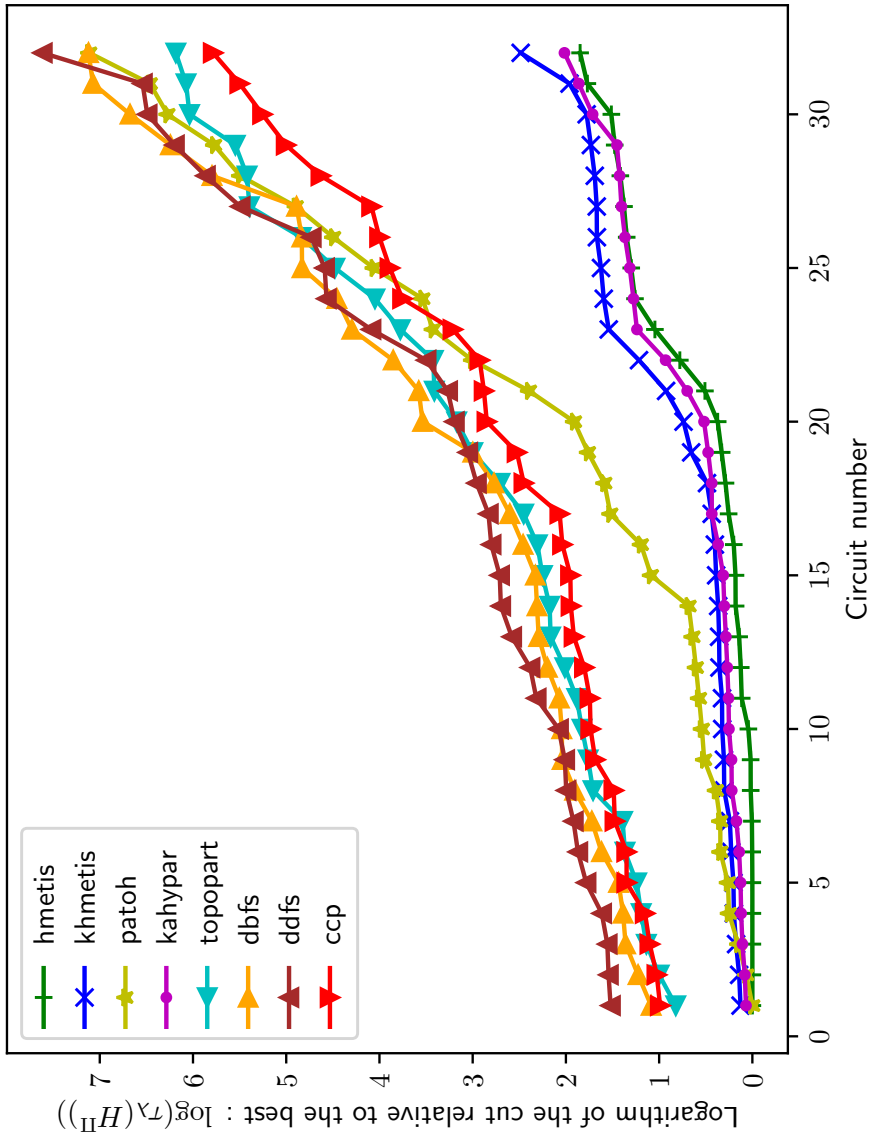


Figure 5.15: Logarithm of connectivity-minus-one cost when mapping onto architecture T6 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and greedy path cost algorithms: DBFS, DDFS and CCP. As for target topology T3, one can see a break between mincut and greedy algorithms. Similarly as on T3, TopoPart behaves like DBFS, DDFS, and CCP, and PaToH produces good results for half of the circuits. DBFS and DDFS produces worst connectivity-minus-one results.

5.6 Conclusion

In this chapter, we introduced two initial partitioning algorithms, DBFS and DDFS, based on graph traversal, for the problem of partitioning with path cost minimization. The aim of these two algorithms is to favor the grouping of critical vertices in order to avoid cuts along critical paths. DBFS explores the red-black hypergraph from DAH to DAH, which is interesting when DAH are highly connected. However, for very sparse DAH, DDFS tends to perform better than DBFS because DDFS explores the hypergraph in depth.

We also presented the CCP algorithm, which is an extension of the cone partitioning algorithm, to tackle the partitioning of red-black hypergraphs with path cost minimization. The structure of a cone provides path grouping properties that are of interest for optimizing the f_p function. In addition, this algorithm locally groups critical components throughout the hypergraph, making CCP more adaptable to various hypergraph topologies than the more specific DBFS and DDFS. However, CCP does not produce an expected number of parts. To get the expected number of parts, CCP must be coupled with a partitioning algorithm.

As integer programming algorithms can be interesting to provide an exact initial partition, we introduced in the previous section our approach based on an integer program. This integer program takes advantage of scheduling constraints to place paths according to the target topology, as opposed to DBFS, DDFS and CCP.

In order to optimize the partition produced by DBFS, DDFS and CCP, in Section 5.4, we made some suggestions for mapping the initial partition.

Experimental results show that the DBFS and DDFS algorithms are relevant and complementary initial partitioning methods, depending on circuit instances and their underlying topologies. These algorithms seem to be a good approach for the prototyping of circuits on a multi-FPGA platform. However, these methods tend to degrade cut size. We presented results of DBFS, DDFS and CCP with min-cut tools on all circuit benchmarks. These results show that routing costs on the FPGA platform has an impact on our algorithms. In fact, for fully connected topologies, CPP yields best results, and DBFS and DDFS produce overall good results, no worse than min-cut tools. These algorithms do a good job in capturing critical paths in placing them in a single part whenever possible, unlike min-cut algorithms that focus solely on cut size.

Results of applying exact solver to our integer program, used as initial partitioning inside a multilevel scheme, show that our approach is better at minimizing f_p than a min-cut partitioning tool, even if it is oriented towards minimizing the sum of the criticality of hyperarcs cut. However, when the hypergraphs are large, the coarsening step over-approximates the paths, resulting in an initial partitioning of lower quality.

Chapter 6

Refinement algorithms

This chapter discusses the refinement algorithms that we studied in the context of this thesis. These algorithms are part of the third step of the multilevel scheme: uncoarsening and refinement. Refinement algorithms aim to improve existing partitions. Given a hypergraph H and a partition Π, refinement algorithms aim to improve Π by moving vertices of the frontier (or halo), *i.e.*, whose hyperedges do not have all their vertices in the same part. In general, refinement algorithms calculate a gain associated with a vertex move. The vertices of highest gain, that is, those which improve the objective function most, are moved first. Some algorithms also accept vertex moves with a negative gain, that is, which degrade the current solution, in order to evade from a local minima and improve the search on the solution space [75, 114]. Other refinement algorithms exist, such as approaches based on computing a minimum separator from a max-flow algorithm. P. Senders *et al.* [168] presented a refinement method based on a max-flow algorithm for the graph partitioning problem. T. Heuer *et al.* [94] adapted this algorithm to hypergraph partitioning. In our hypergraph partitioning context, computing a separator of minimum cut does not necessarily improve the solution quality. We are interested in refinement algorithms based on local search, such as KL [114] and FM [75], because these algorithms have proven their effectiveness and ability to adapt to different objective functions. KL, introduced by N. W. Kerninghan and S. Lin, explained in Section 6.1.1, and FM, introduced by C. M. Fiduccia and R. M. Mattheyses, is described in Section 6.1.2. These two approaches inspired our Delay K-way FM refinement algorithm, presented in Section 6.1.4. Some sections of this chapter are based on our published work [160].

6.1 Refinement algorithms

In this section, we outline the methods that inspired our refinement algorithm for the problem of partitioning a red-black hypergraph. The algorithm presented here are suited for (hyper)graphs with a single weight on their vertices and hyperarcs, and can be extended to a vector of weights with minimal adaptations.

6.1.1 The Kerninghan - Lin Algorithm

The algorithm proposed by N. W. Kerninghan and S. Lin [114] (or KL), described as Algorithm 9, concerns the graph bisection problem. KL aims at improving a bipartition by moving vertices from one side of the cut to the other. Given a bipartition of the graph $\Pi = \{\pi_0, \pi_1\}$, as shown in Figure 6.1, the goal of the algorithm is to find subsets of misplaced vertices $A \subset \pi_0$ and $B \subset \pi_1$, such that, by swapping A into π_1 and B into π_0, we get $\pi'_0 = (\pi_0 \setminus A) \cup B$ and $\pi'_1 = (\pi_1 \setminus B) \cup A$, and the updated partition $\Pi' = \{\pi'_0, \pi'_1\}$ has a better cost than the initial partition

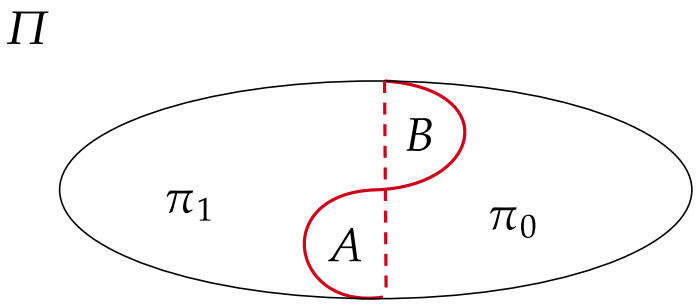


Figure 6.1: Example of partition status after the refinement process. Here, the plain and the dotted lines are the current and desired frontier of the partition, respectively.

Π. In the KL algorithm, each vertex is associated with an integer value $\text{gain}^1_{KL}(u)$ which evaluates the impact on the cost function of the movement of the vertex from its current part to the other part. The gain for moving vertex u from part π_0 to part π_1 is defined by:

$$\text{gain}^1_{KL}(u) \stackrel{\text{def}}{=} \sum_{v \in \Gamma(u) \cap \pi_1} W_e(u,v) - \sum_{v \in \Gamma(u) \cap \pi_0} W_e(u,v) \ . \tag{6.1}$$

The first term evaluates the sum of the weights of the formerly cut edges that will no longer be cut after the move, while the second term evaluates the cost of the edges that will be cut by the move. Since the KL algorithm is applied to the graph bisection problem, *i.e.*, a balanced bipartition, moving a vertex from part π_0 to part π_1 implies the move of another vertex from part π_1 to π_0, so as to maintain the balance of the partition. Therefore, one can compute a gain per pair of two swapped vertices u, v, defined by:

$$\text{gain}^2_{KL}(u,v) \stackrel{\text{def}}{=} \text{gain}^1_{KL}(u) + \text{gain}^1_{KL}(v) - 2 \times W_e(u,v) \ . \tag{6.2}$$

The KL Algorithm 9 performs a series of refinement passes on the current bisection. Each pass computes the gain for each edge in the cocycle of the bisection. Then, a selection of the best couple of vertices to swap is made. When a pair of vertices v_0 and v_1 is selected, it is "locked", to avoid looping infinitely on the same vertices, and the gains of its neighbors are updated, to simulate their move.

Algorithm 9 Kernighan-Lin Refinement Algorithm (KL)

Require: $G = (V, E), W_e$
Ensure: Bisection $\Pi = \{\pi_0, \pi_1\}$

repeat
  for $(v_0, v_1) \in \omega(\Pi)$ **do**
    $\text{gain}^2_{KL}(v_0, v_1) \leftarrow \text{gain}^1_{KL}(v_0) + \text{gain}^1_{KL}(v_1) - 2 \times W_e(v_0, v_1)$ ▷ The gain is set for each pair of neighbor across the cut
  end for
  nlock $\leftarrow 0$
  is_locked $\leftarrow [False]^{|V|}$
  moves $\leftarrow []$
  while nlock $< |V|$ **do**
    $v_0, v_1 \leftarrow \text{gain}^2_{KL}.get(0)$ ▷ Return the first couple of vertices
    moves.append$((v_0, v_1))$
    is_locked$(v_0) \leftarrow True$
    is_locked$(v_1) \leftarrow True$
    nlock $\leftarrow$ nlock $+ 2$
    for $v'_0 \in \Gamma(v_0)$ **do** ▷ For each unlocked neighbor, the gain is updated with its neighbors
      if ¬is_locked(v'_0) **then**
        for $v''_0 \in \Gamma(v'_0)$ **do**
          if ¬is_locked(v''_0) **then**
            $\text{gain}^2_{KL}(v''_0, v'_0) \leftarrow \text{gain}^1_{KL}(v''_0) + \text{gain}^1_{KL}(v'_0) - 2 \times W_e(v'_0, v''_0)$
          end if
        end for
      end if
    end for
    for $v'_1 \in \Gamma(v_1)$ **do** ▷ For each unlocked neighbor, the gain is updated with its neighbors
      if ¬is_locked(v'_1) **then**
        for $v''_1 \in \Gamma(v'_1)$ **do**
          if ¬is_locked(v''_1) **then**
            $\text{gain}^2_{KL}(v''_1, v'_1) \leftarrow \text{gain}^1_{KL}(v''_1) + \text{gain}^1_{KL}(v'_1) - 2 \times W_e(v'_1, v''_1)$
          end if
        end for
      end if
    end for
  end while
  $i^* \leftarrow -1$ ▷ Initial value for the index of "good" moves, -1 implies no moves

34: gain$^* \leftarrow -1$ ▷ Initial value for the best gain, *i.e.*, -1 enforce to choose the first move with a positive gain
35: gain$' \leftarrow 0$
36: **for** $i \in |\text{moves}|$ **do** ▷ Each move is simulated
37: gain$' \leftarrow$ gain$' +$ gain(moves(i))
38: **if** gain$^* <$ gain$'$ **then**
39: i$^* \leftarrow i$
40: gain$^* \leftarrow$ gain$'$ ▷ The current best solution is updated
41: **end if**
42: **end for**
43: **if** gain$^* > 0$ **then**
44: **for** $i \in \{0, i^*\}$ **do**
45: $v_0, v_1 \leftarrow$ moves(i)
46: swap(v_0, v_1) ▷ Each selected moves to obtain the best gain is applied
47: **end for**
48: **end if**
49: **until** gain$^* > 0$
50: **return** $\Pi' = \{\pi_0, \pi_1\}$

At the end of the loop, all swaps have been simulated, *i.e.*, all vertices of π_0 are now in π_1, and all vertices in π_1 are in π_0. Finally, each swap is processed in the order in which it was inserted, *i.e.*, by descending order of gain. The gain can be negative if the swap increases the size of the bisection. Each gain is added to a sum of gains which measures the quality of the sequence of exchanges between $[0, i^*]$, where i^* is the index of the last swap for which the sum of gains is maximized. The search continues for all swaps, so as to overcome a local maximum. Finally, the swaps between $[0, i^*]$ are applied, and the procedure is repeated until the cost of a pass is positive. The worst-case complexity of the algorithm is in $O(V^3)$ time [114]. However, S. Dutt *et al.* [65] presented an improvement in execution time, in $O(|E| \max(\log(|V|), \Delta))$.

6.1.2 The Fiduccia-Mattheyses Algorithm (FM)

The well-known FM refinement algorithm [75], devised by C. M. Fiduccia and R. M. Mattheyses, addresses the bipartitioning problem for hypergraphs. Starting from a bi-partition $\Pi = \{\pi_0, \pi_1\}$, the algorithm performs passes, *i.e.*, sets of vertex moves from one part to the other. Let us recall that, in this chapter, we are considering hypergraphs with unit weights. As with the KL algorithm, the FM algorithm will evaluate a moving gain per vertex. Given a vertex $v \in \pi_0$, the gain

function $\text{gain}_{\text{FM}}(v, \pi_1)$ associated with the moving of v to part π_1 is defined by:

$$\text{gain}_{\text{FM}}(v, \pi_1) \stackrel{\text{def}}{=} \sum_{e \in E, \forall u \neq v \in e \cap \pi_1 \ s.t. \ v \in e} W_e(e) - \sum_{e \in E, \forall u \neq v \in e \cap \pi_0 \ s.t. \ v \in e} W_e(e) \ . \quad (6.3)$$

The gain of a move can be negative or positive. Let $\lambda(v) = |\{e | v \in e, e \in E\}|$ be the connectivity of vertex v, *i.e.*, the number of hyperedges connected to this vertex. In the electrical engineering literature, vertices are called "pins", hyperedges are called "nets", and the connectivity value $\lambda(v)$ is the number of nets containing some pin. Let $\Lambda_V = \max\{\lambda(v), v \in V\}$ be the maximum degree in the hypergraph. It is possible to bound the gain function for any vertex v by $[-\lambda(v), +\lambda(v)]$ and in the general case, by $[-\Lambda_V, +\Lambda_V]$. Consider the following example, with all hyperedges connected to v have all their vertices also in part π. If v moves to another part, then $\lambda(v)$ new hyperedges will be cut. This is the worst case, with a gain equal to $-\lambda(v)$. In the opposite case, when v is in one part and all its neighbors, *i.e.*, the vertices contained in hyperedges incident to v, are in another part π, moving v into part π will yield a gain of $+\lambda(v)$. Vertices are chosen each time by taking the vertex of best gain. If a partition is considered as unbalanced, moves that rebalance it are allowed, even if their gain is negative. Allowing moves with negative gain may allow the algorithm to escape from local minima.

At the end of the pass, the balanced partition with the best cost is selected. As with the KL algorithm, each vertex can only be moved only once during each refinement pass. In the FM algorithm for non-weighted hypergraphs, a bucket-list (BL) data structure stores vertex gains. This data structure maintains two arrays of size $2\Lambda_V$, in which each cell i is linked to a doubly-linked list containing the vertices of gain i. The first array stores the vertices that are susceptible to move from π_0 to π_1, and the second array stores the vertices that are susceptible to move from π_1 to π_0. If a vertex v is connected to gain x of $\text{BL}(\pi_0)$, its gain for moving to π_1 is x. The authors show in their work [75] that the run-time complexity of the FM algorithm is in $O(\Lambda_V)$ operations per pass using this data structure.

6.1.3 K-way Fiduccia-Mattheyses (KFM)

In their version of the refinement algorithm, C. M. Fiduccia and R. M. Mattheyses presented an algorithm designed for bipartitioning. In the k-way partitioning context, either recursive bipartitioning must be used, or the FM algorithm must be adapted to k-way partitioning as in L. A. Sanchis [167]. Recursive bipartitioning consists in dividing the hypergraph into two parts, and repeating the procedure until the hypergraph is divided into k parts. Several algorithms based on recursive bipartitioning have been proposed [33, 34, 56, 107, 110, 149, 174], mainly because

of their lower complexity and ease of implementation. The k-way refinement algorithm generally follows the same pattern as the 2-way algorithm described in the previous section. Several works [6, 16, 112, 167] have proposed extensions of the FM refinement algorithm to KFM, "K" standing for k-way partitioning. Readers interested in the different implementations of variants of the Fiduccia and Mattheyses' algorithm can e.g., refer to the work of Ü. Çatalyürek *et al.* [35].

6.1.4 Delay K-partitioning Fiduccia-Mattheyses (DKFM)

As seen in previous sections, the FM refinement algorithm is widely used in hypergraph partitioning methods. In this subsection, we present our Delay K-FM (DKFM) algorithm, an extended version of the KFM refinement algorithm for delay-minimization k-way partitioning. This algorithm is adapted to the red-black hypergraph partitioning problem and aims at minimizing both the impact of partitioning and of routing on critical paths.

Let Π be a partition of H; the gain function for some candidate partition Π', in which some vertex may be moved to a different part, is defined as:

$$\text{gain}_{\text{DKFM}}(\Pi, \Pi', H) \stackrel{\text{def}}{=} d^{\Pi}_{max}(H) - d^{\Pi'}_{max}(H) \ . \tag{6.4}$$

When calculating the gain, the movement of a vertex from one part to another is simulated, to calculate the effect of this movement on the value of the critical path. The recalculation of the critical path has a complexity in $O(|V|)$ time. Indeed, if the vertex is red, the algorithm computes the critical path in all concerned DAHs. In the worst case, the red vertex connects to all DAHs, which explains this maximum complexity in $O(|V|)$. When a vertex is moved from one part to another, the critical path update can occur throughout the DAH. For example, in Figure 6.2, the critical path of the Π partition is p_a. The cost of the Π partition is therefore $d^{\Pi}_{\max}(H) = d(p_a)$. Moving vertex v to part π_0 seems to be a good choice, as it reduces the cost of the critical path. Π' is the result of moving v to part π_0. Since p_a is no longer routed to π_1, path p'_a is such that $d(p'_a) < d(p_b)$. Unless we calculate the effect of partition Π' on the red-black hypergraph $d^{\Pi'}_{\max}(H)$, one cannot obtain the value of the new critical path locally, or by a local calculation on the neighborhood of v, or of p_a.

The problem addressed in classical partitioning algorithms is the minimization of the cut size f_c, or connectivity cost f_λ. These functions are defined by a sum of local hyperedges cut/connectivity cost. Hence, if a hyperedge is cut, then, it is accounted for in the sum of cut hyperedges. Hence, because of the associativity property of a sum, if a cost of one hyperedge change, f_c and f_λ cost functions do not need to recompute the cost of each hyperedge to be evaluated.

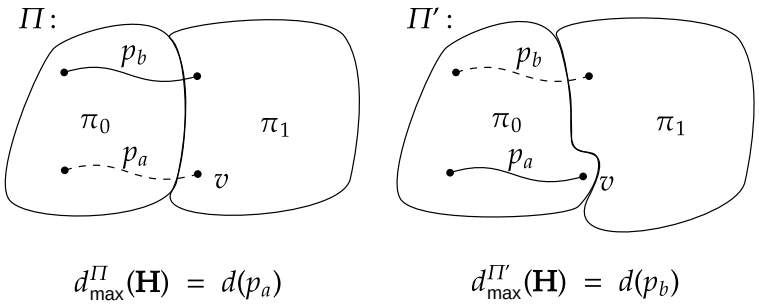


Figure 6.2: Illustration of what is happening when a vertex is moved from part π_1 to π_0. In this example, p_b, a path in the hypergraph, may become the new critical path.

Consequently, a refinement algorithm for the problem addressed in this dissertation has basically a higher run-time complexity than min-cut refinement algorithms. There are two parts of our algorithm in which we perform the evaluation of the critical path; when DKFM computes the gains of vertices in the frontier, and, when DKFM moves a vertex to another part.

For the first case, when calculating gains, a way to reduce the number of critical path evaluations is to reduce the number of evaluated vertices. Hence, we propose to compute gains for a subset of randomly selected vertex candidates. Indeed, the algorithm will perform a parameterized number of insertions before moves. Note that the probability of processing the same vertex twice is $1/|\hbar|^2$, with $|\hbar|$ being the number of vertices in the frontier.

For the second case, when DKFM moves a vertex, we study some ways to avoid recalculating the propagation of the cut overhead along all impacted paths for each candidate move. In Chapter 4, we have defined the notion of cut capacity, *i.e.*, a number of cuts which will not degrade the maximum path cost. Cut capacity is a local evaluation based on local criticality between two vertices, *i.e.*, the length of the longest path traversing these two vertices. However, cut capacity can be used to measure the probability of critical path modification. Hence, the definition of the cut capacity should be extended to a *degradation capacity*, accounting for other parameters in order to tune its importance.

Let ς be a degradation capacity associated with path p. This capacity is relative to the latest computed critical path and the cut cost D_{ij}. We define ς for each pair of connected vertices (u, v), with $d^{\Pi}_{\max}(u, v)$, representing the length of the local critical path traversing u and v. We define ς as:

$$\varsigma(\mathbf{H}^{\Pi}, u, v) \stackrel{\text{def}}{=} \rho(\overline{d^{\Pi}_{\max}} - d^{\Pi}_{\max}(u, v))/D_{\Pi[u]\Pi[v]} \ , \tag{6.5}$$

with u and v being the source and sink vertices of some hyperarc, and ρ a cut tolerance. A cut tolerance is a parameter which tunes the degradation capacity ς. Hence, if ρ is equal to zero, each degradation capacity is equal to zero. In this case, the critical path is evaluated for each modification of the partition. When $\rho > 0$, $d_{\max}(H)$ will be updated if $\varsigma(H^{\Pi}, u, v) \leq 0$. The value of ρ drives the quality of the gain computation during a DKFM pass.

The description of a refinement pass is provided as Algorithm 10. The first step consists in randomly selecting a part from which a vertex can be moved without overflowing the part size. Consequently, the algorithm can rebalance unbalanced partitions at the start of processing, by encouraging the movement of vertices from overloaded to underloaded partitions, even if the resulting gain is negative.

Once at line 2, part k' is selected and the vertex to move is retrieved from the list of moves associated with part k'. The vertex with the best gain is selected using an array that stores the best gain for each part. The following lines, from line 3 to line 9, move the vertex v from k to k'. The new critical path associated with the new partition i' is calculated at line 9. If the new part k' exceeds the maximum capacity $M_{k'}$, then part k' is removed from the candidate parts, and no vertex is added to part k' until the capacity constraint is met again. Finally, we update the current best solution if the new critical path length is lower than the previous best solution. Each vertex that is moved is locked, to avoid multiple moves or back-and-forth movement.

The `for` loop at line 21 performs the gain update processing on the neighborhood of v. We only calculate the gain of a move from v' neighbors to the new part of v, k'. If the neighbors of v are no longer connected to the old part of v, part k, then k is removed from the set of neighboring parts $B(v')$. Moves of neighbors v' that are no longer connected to part k are also removed. Finally, we calculate the gain of neighboring vertices for a move to k'. If vertices were not connected to part k', then part k' is added to set $B(v'')$ and the gain of a move of v'' to k' is calculated. We will not worry about the other parts, as they have already been calculated. In the case where the critical path value has changed, *i.e.*, in the case of a deterioration or an improvement due to the moving of v in part k', the critical path value has to be updated according to the capacity degradation ς. It is assumed that the gains will still be locally accurate and will reduce the local critical path degradation.

In addition, the DKFM algorithm is called at each level during the uncoarsening stage. As a result, the gains are recomputed for each level. If a critical path changes, it will be processed at the next level. The complete algorithm in pseudo-code is shown as Algorithm 11. The data structure used in the DKFM algorithm

Algorithm 10 Delay K-way Fiduccia-Mattheyses local search algorithm (one pass)

Require: $H = (V, A, W_v, W_e)$ a red-black hypergraph, $B(v)$ a function which indicates the parts in the neighborhood of a vertex v, k number of part, ς the degradation capacity, ρ a cut tolerance, ϵ a imbalance ratio, `moves` a table that stores vertex moves, `is_locked` a boolean array

Ensure: Π a k-partition of **H**

$i \leftarrow$ `random`$(0, |$`part_candidat`$|)$
$k' \leftarrow$ `part_candidat`$[i]$
$v_{k'}, \varsigma_v \leftarrow$ `moves`$[k']$[`best_gain`$[k']]$ ▷ $v_{k'}$ is current best vertex to move in part k'
$v \leftarrow v_{k'}$ ▷ Get the index of vertex v
$k \leftarrow \Pi(v)$ ▷ Return the current part containing v
$\Pi' \leftarrow \Pi \setminus \{\pi_k, \pi_{k'}\}$
$\pi_k \leftarrow \pi_k \setminus \{v\}$
$\pi_{k'} \leftarrow \pi_{k'} \cup \{v\}$
$\Pi' \leftarrow \Pi' \cup \pi_{k'}, \Pi' \leftarrow \Pi' \cup \pi_k$
$p'_{\max} \leftarrow d^{\Pi'}_{\max}(\mathbf{H})$ ▷ Partition Π' and the length of the critical path are updated
if $|\pi_{k'}| > M_{k'}$ **then**
    `part_candidat`.remove(k')
end if
if $k \notin$ `part_candidat` $\wedge\ |\pi_k| < M_k$ **then**
    `part_candidat`.add(k) ▷ Each part that does not respect the constraint size cannot receive another vertex
end if
if $p'_{\max} < p_{\max}$ **then**
    $p_{\max} \leftarrow p'_{\max}$
    $\Pi \leftarrow \Pi'$ ▷ Current best solution is updated
end if
`is_locked`$[v] \leftarrow$ `True` ▷ Each vertex is moved only once per pass
for $v' \in \Gamma(v)$ **do**
    if $\neg$`is_locked`$[v'] \wedge \varsigma_v \leq 0$ **then**
        for $v'' \neq v \in \Gamma(v')$ **do**
            `is_connected` $\leftarrow$ False
            if $\Pi(v'') = k$ **then** ▷ If vertex v'' is in the old part of v: k, v'' is flagged
                `is_connected` $\leftarrow$ `True`
            end if
        end for
        if $\neg$`is_connected` **then** ▷ If v' has no neighbors in part k

31: $B(v')$.remove(k) ▷ The move of v' to part k is removed
32: remove(`vertex_pointer`, v', k)
33: **end if**
34: **if** $\Pi(v') \neq \Pi(v)$ **then**
35: $k \leftarrow \Pi(v')$ ▷ Here, k is the new part of v
36: $\Pi' \leftarrow \Pi \setminus \{\pi_k, \pi_{k'}\}$
37: $\pi_k \leftarrow \pi_k \setminus \{v'\}$
38: $\pi_{k'} \leftarrow \pi_{k'} \cup \{v'\}$
39: $\Pi' \leftarrow \Pi' \cup \pi_{k'}, \Pi' \leftarrow \Pi' \cup \pi_k$
40: gain $\leftarrow p_{\max} - d^{\Pi'}_{\max}(\mathbf{H})$
41: $\varsigma_{v'} \leftarrow \rho(p_{\max}) - r(v')/D_{kk'}$
42: **if** $|B(v')| = 0 \wedge v' \notin \pi_{k'}$ **then** ▷ v' goes to the halo
43: $B(v')$.add(k') ▷ k' goes to the neighbors parts of v'
44: `is_locked`[v'].add(v')
45: **else**
46: remove(`vertex_pointer`, v', k) ▷ Delete the old gain of moving v' to k'
47: **end if**
48: `moves`[k'][gain].insert($v', \varsigma_{v'}$) ▷ Insert the new gain of moving v' to k'
49: **end if**
50: **end if**
51: **end for**
52: **return** Π

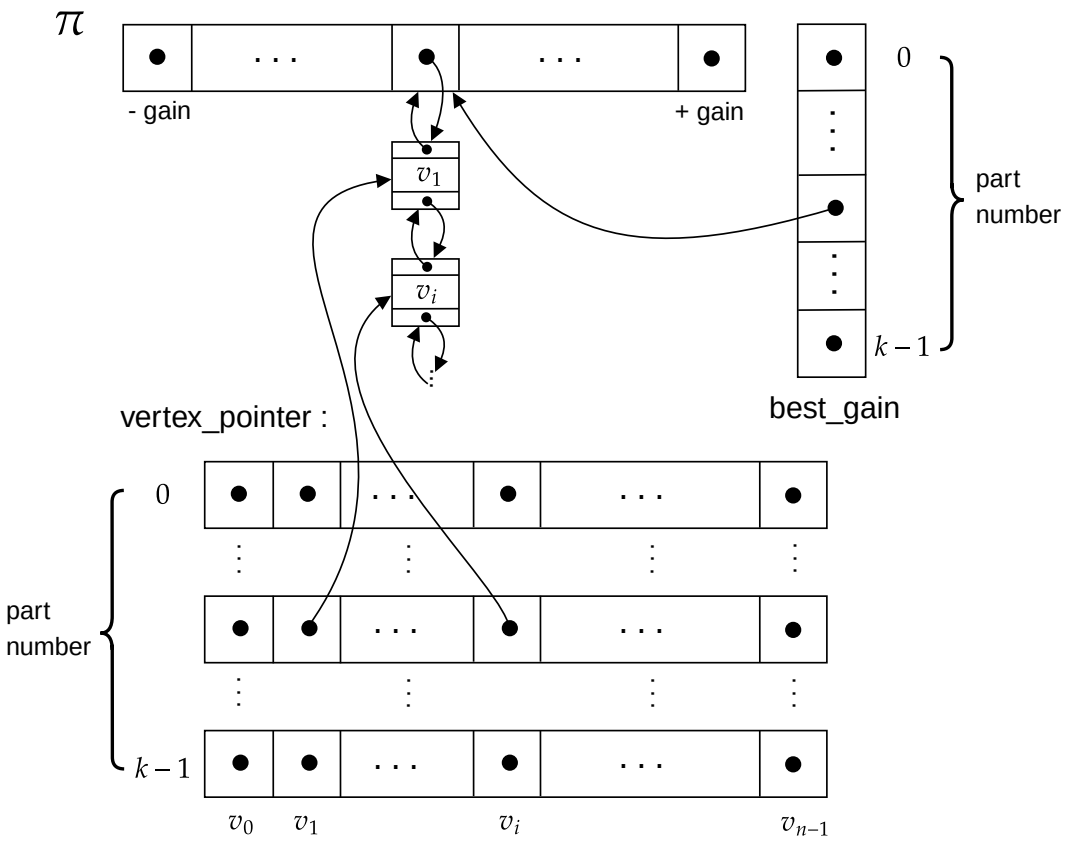


Figure 6.3: Implementation of data structures for the DKFM algorithm. `vertex_pointer` arrays can be sparse; hence, is it possible to optimize the memory footprint, which is in $O(k \times n)$ space, with k being the number of parts and n the number of vertices.

is similar to that used in the FM algorithm. The data structure consists of a gain array, where each cell represents the value of a gain, bounded by the minimum gain and the maximum gain. Each cell in the array contains a doubly-linked list of vertices to be moved to the said part, with a gain corresponding to the cell index in the array. Such an array is constructed for each part π_k. In addition to this structure, the array `vertex_pointer` associates the vertices with their pointers in the doubly-linked list, to find them more efficiently. The `best_gain` array stores a pointer to the best gain for each part. The `best_gain` array gives access to the best moves in $O(1)$ time. An example of this data structure for some part π_k is shown in Figure 6.3.

Lemma 6.1.1. *One pass of the Delay K-way Fiduccia-Mattheyses local search Algorithm 10 runs in $O(|V|^2)$, with $|V|$ being the number of vertices.*

Proof. The algorithm chooses a part in which to make a move. A vertex of best gain is selected, and a gain update is calculated for its neighborhood. The algorithm performs assignations and reads from lines 1 to 9 in $O(1)$ time. At line 10, the computation of the new value of the critical path applies to the entire red-black hypergraph, in the worst case. This results in a computation time in $O(|V|)$. The `for` loop at line 22 iterates over the entire neighborhood of the moved vertex, v. In the worst case, $|\Gamma(v)| = |V| - 1$. In this loop, we perform a second traversal of the neighbors of v and a critical path calculation at line 43, evaluating the new gain of each vertex. Since the calculation of the critical path and the traversal of the neighbors are, in the worst case, in $O(|V|)$ time, and $|\Gamma(v)| = |\Delta| - 1$, we get a complexity in $O(\Delta|V|)$ time. The operations of reading, inserting, and deleting the structure of the moves are performed in constant time using a doubly-linked list. □

Lemma 6.1.2. *The Delay K-way Fiduccia-Mattheyses local search Algorithm 11 runs in $O(|V|^2 log_2(|V|))$ time, with $|V|$ being the number of vertices.*

Proof. Algorithm 11 applies several refinement steps to an initial partition provided as a parameter. At line 1, the algorithm calculates the halo λ, *i.e.*, the set of vertices present at the frontier of the partition. In the worst case, this calculation is done in $O(|V|^2)$ time if each vertex is connected to all the other vertices. The processing between lines 4 and 10 is performed in $O(k|\lambda_\Pi| \times |V|)$ time. Note that the insertion into the list of moves of part k performed at the index corresponding to its gain, hence, is performed in constant time, because it is an insertion into a doubly-linked list. In the worst case, the processing between lines 4 and 10 takes $O(k \times |V|^2)$ time. The computation of the best gain for each part at line 11 is completed in $O(k \times |V|)$ time. Candidate parts are computed in $O(|V|)$ time, by adding the weight of each vertex corresponding to its gain. Finally, the main loop at line

Algorithm 11 Delay K-way Fiduccia-Mattheyses local search algorithm (pseudo-code)

Require: $H = (V, A)$ a red-black hypergraph, W_v vertex weights, k a part number, ς a capacity degradation, ρ a tolerance ratio, ϵ a imbalance ratio, N the number of passes

Ensure: Π a k-partition of H

Compute the `frontier` λ_Π of partition Π
Initialize the`moves`arrays (each for one part)
Initialize the is_locked array
for $v \in \lambda_\Pi$ **do**
    Compute the neighbor parts of v, and put them in $B(v)$
    for $k \in B(v)$ **do**
        Compute the gain of moving v to part k
        Insert the gain in `moves`$[k]$[gain]
    end for
end for
Compute the best move for each part
Compute the candidate parts ▷ *i.e.*, parts that respect the capacity constraint
`nb_passes` $\leftarrow 0$
while `nb_passes` $< N$ **do**
    Call DKFM pass
end while
return Π

13 makes a number of calls to the DKFM pass described as Algorithm 10. The complexity of this pass is in $O(|V|^2)$ time, as seen in Section 6.1.1. Since the `while` loop is repeated N times, the complexity for the Delay K-way Fiduccia-Mattheyses local search Algorithm 11 is in $O(|V|^2 + |V|^2 + N \times |V|^2)$ time. When N is set in $O(\log_2(|V|))$, the complexity of the `while` loop at line 13 is in $O(|V|^2 \log_2(|V|))$ time. Then, for N in $O(\log_2(|V|))$, the total complexity for the Delay K-way Fiduccia-Mattheyses local search Algorithm 11 is in $O(|V|^2 + |V|^2 + |V|^2 \log_2(|V|))$, that is, in $O(|V|^2 \log_2(|V|))$ time. □

The dedicated data structure, first introduced by C. M. Fiduccia and R. M. Mattheyses [75] and extended to the DKFM algorithm, can have a high memory footprint for large red-black hypergraphs. As presented in the thesis of S. Schlag and previous work [146, 173], priority queues can be used to store best moves for each part. Consequently, we will also use priority queues to improve the complexity of our DKFM algorithm. Let us study the complexity of DKFM with priority queue data structures, instead of the classic FM data structure.

Lemma 6.1.3. *The Delay K-way Fiduccia-Mattheyses local search Algorithm 11 runs in $O(|V|^2 log_2(|V|))$ time, with $|V|$ being the number of vertices, and k the number of parts, with a priority queue data structure for vertex gains.*

Proof. The algorithm applies several refinement steps to an initial partition given as a parameter. First, the algorithm computes the halo λ_Π, *i.e.*, the set of vertices located at the frontier of the partition. In the worst case, this calculation is performed in $O(|V|^2)$ time, each vertex connected to all the other vertices. Then, the gain for each vertex in the halo is computed in $O(k \times |V|)$ time, because the computation of the new critical path in $O(|V|)$ time has to be performed for each connected part. In the worst case, the halo contains all vertices and each vertex is connected to each part. Then, as a priority queue is used instead of doubly-linked list, the insertion of a vertex is performed in $\log_2(V)$ time. A heap data structure is used to represent the priority queues. Then, the worst time complexity for the calculation of the vertex gain is in $O(k \times |V| \times (|V| + \log_2(V))) = O(k \times |V|^2)$. At this step, the complexity is similar to the previous version, because the computation of the critical path has a higher complexity than gain insertion. Each best move is present in the head of each priority queues.

The second part of the algorithm consists of N vertex movements across the frontier of the partition. For each movement, a vertex is dequeued from a selected candidate part. The selection of a candidate part is made in $O(k)$ time and the dequeued operation in $O(\log_2(V))$ time. Then, after each move, the gains of neighbor vertices have to be updated. In the worst case, the number of neighbors is $|V|$, that is, the worst time complexity for updating gains is in $O(|V| \times (V + \log_2(|V|)))$. Since the `while` loop is repeated N times, the complexity for the

Delay K-way Fiduccia-Mattheyses local search Algorithm 11 is in $O(|V|^2 + |V|^2 + N \times |V|^2)$ time. When N is in $O(\log_2(|V|))$, the complexity for the `while` loop on line 13 is in $O(|V|^2 \log_2(|V|))$ time. Then, since N is in $O(\log_2(|V|))$, the total complexity for the Delay K-way Fiduccia-Mattheyses local search Algorithm 11 is in $O(|V|^2 + |V|^2 + |V|^2 \log_2(|V|))$, that is, in $O(|V|^2 \log_2(|V|))$ time when using a priority queue data structure instead of doubly-linked lists for part moves. □

In the previous Lemma 6.1.3, we show that the use of priority queues instead of doubly-linked lists does not degrade the complexity of DKFM in the worst case. This is specific to the problem of path cost minimization, because the computation of gain forces the evaluation of the critical path for each partition. As the calculation of the critical path is performed with a higher complexity, the management of the priority queue is absorbed in the worst case complexity analysis.

6.2 Experimental results

This section presents the results for a version of DKFM in which all possible gains are computed on the circuits presented in Chapter 3. That is, the parameter ρ is set to 0, the critical path is calculated after each movement and the evaluation of f_p is made for each gain. Hence we refer to this version of DKFM as : "DKFM", and the version of DKFM with optimization presented previously, as : "DKFMFAST". We will present the results of both, DKFM and its fastest version called DKFMFAST.

6.2.1 Methodology

In Chapter 2, we presented existing approaches based on pre- and post-processes using min-cut solvers as a main algorithm for partitioning. To the best of our knowledge, there is no publicly available tool to tackle our problem; hence, the scientific community has developed such procedures in combination with existing min-cut tools to handle path cost. Following these approaches, we investigated the efficiency of a refinement algorithm dedicated for our problem, combined with existing min-cut tools.

Chronologically, the DKFM algorithm is the first we have developed in the course of this thesis. For this reason, we have created a benchmark in which DKFM is a post-processing procedure of a mincut tool. However, the pre- and post-processes presented in the state of the art are not publicly available. This makes it difficult to compare against them. Hence, the aim of this benchmark is to measure the relevance of using DKFM as a post-processing method.

To test the DKFM refinement algorithm, we first ran hMetis, PaToH, khMetis, and kKaHyPar on the weighted hypergraph instances. We use the r^* weighting

scheme introduced in Chapter 4 to drive min-cut to avoid cutting critical hyperarcs. After this partitioning step, we applied our DKFM algorithm as a post-processing to these outputs, to study the path-cost improvement. We set a time limit of 400s to our DKFM and DKFMFAST algorithms, the parameter ρ to 0, and the maximum number of DKFM moves to 20% of the number of vertices.

We set the hMetis and khMetis balance parameter to 5%, and the number of runs to 10. For the coarsening step, we chose the heavy edge matching strategies to take advantage of our weighting based on vertex criticality. We chose to use PaToH in its default setting to see whether it could produce acceptable solutions in combination with DKFM, because computation time is a critical aspect of industrial-size circuit prototyping. We set the vertex visit order (VO) to *maximum net size sorted* mode and matching type (MT) to *heavy connectivity clustering* algorithm. This vertex visit order favors the processing of critical hyperarcs first. In addition, we set the partitioning algorithm (PA) to *greedy hypergraph growing with max net*. We set kKaHyPar, for a connectivity minus one objective with same parameters indicated on KaHyPar webpage [173][1].

6.2.2 Results of DKFM on critical path degradation

This subsection presents results of our DKFM refinement algorithm applied to each partition computed by oriented min-cut algorithms, on the following six target topologies defined in Chapter 3.

Figure 6.4 and Figure 6.5 present the results of our DKFM refinement algorithm and show that the algorithm succeeds in refining the partitions given as parameters with only 20% of the possible moves made. Since cut minimization algorithms do not address the problem of minimizing critical path degradation and do not take into account the target topology, our algorithm has some leeway to optimize partitions. However, we find that for some partitions, especially those generated by TopoPart and PaToH, DKFM has difficulty improving them. As a result, DKFM may have difficulty escaping a local minimum. Remember that DKFM is applied only once and performs a maximum number of moves less than of 20% of the number of vertices. Note that in the T2 topology, the partitions created by the cut minimization algorithms suffer a much higher routing penalty than in T1. In fact, T2 has no connection between parts 0 and 3, which can result in a very high routing penalty for paths between parts 0 and 3.

Results on target topologies T4 and T5 composed by 8 parts can be consulted in Figure 6.7 and Figure 6.8. Results on fully target topologies T3 and T6 can be consulted in Figure 6.6 and Figure 6.9.

[1] https://kahypar.org/

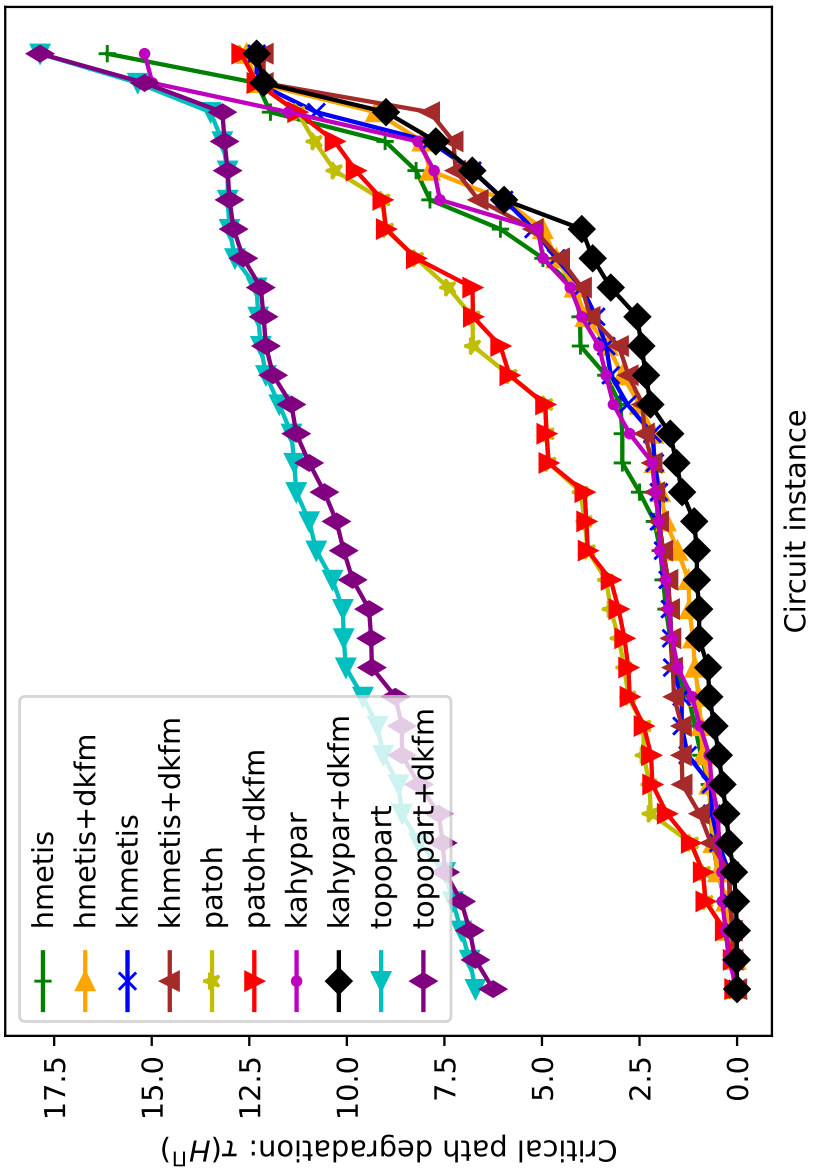


Figure 6.4: Results of improvement produced by DKFM applied to partitions computed by hMetis, khMetis, PaToH, KaHyPar, and TopoPart onto T1. For this topology, KaHyPar+DKFM seems to be better than the others. However, khMetis+DKFM curves is under KaHyPar+DKFM for some instances. Note that DKFM evidences difficulties to improve partitions produced by TopoPart and PaToH.

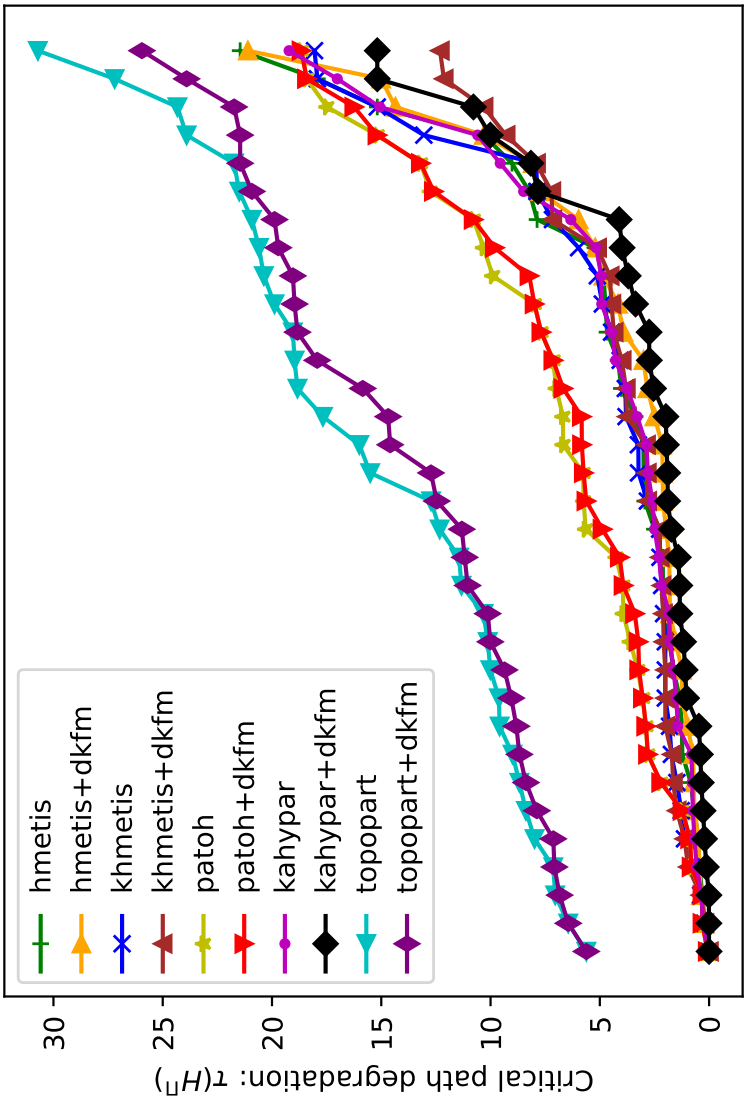


Figure 6.5: Results of improvement produced by DKFM applied to partitions computed by hMetis, khMetis, PaToH, KaHyPar, and TopoPart onto T2. For this topology, KaHyPar+DKFM seems to be better than the others except from index 25 where khMetis+DKFM produced less degradation. DKFM shows difficulties to reduce the cost of partition produced by TopoPart. Note that hMetis and hMetis+DKFM have better results for 75% of circuits compared to PaToH and PaToH+DKFM but hMetis and hMetis+DKFM produced higher maximal possible degradations than PaToH and PaToH+DKFM.

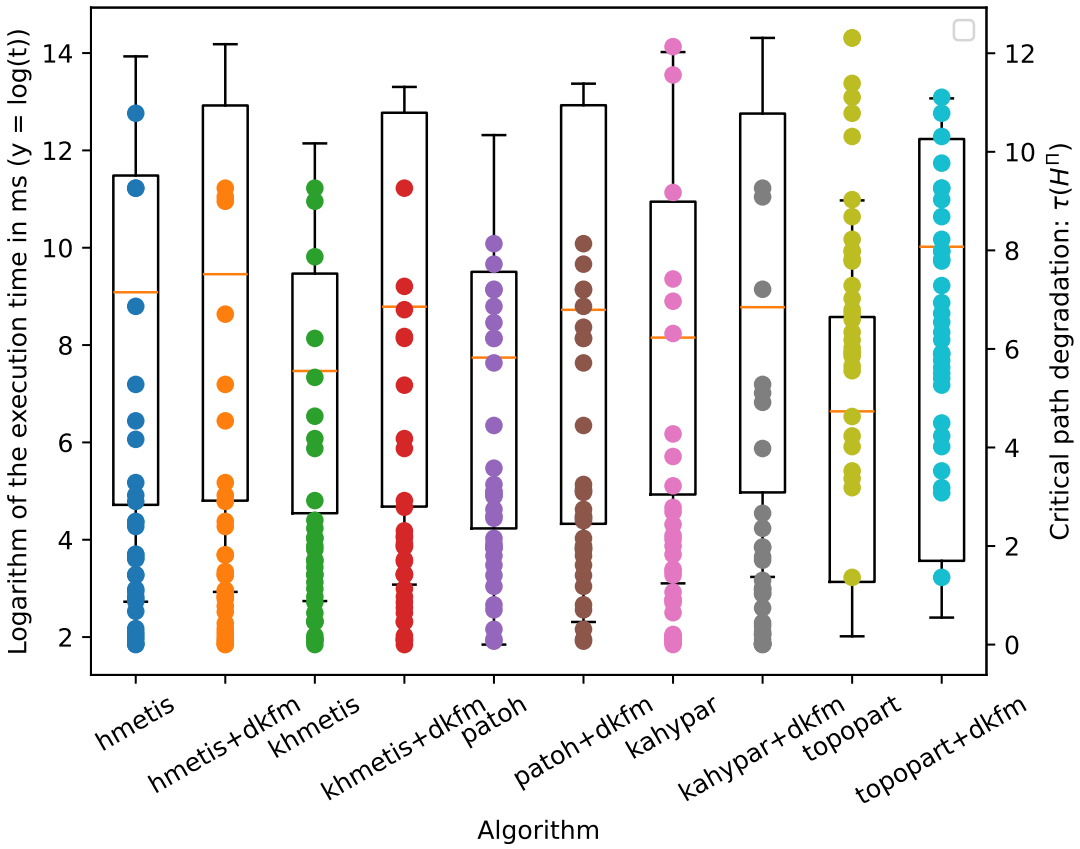


Figure 6.6: Results of improvement produced by DKFM applied to partition computed by hMetis, khMetis, PaToH, KaHyPar, and TopoPart onto fully connected T3 composed of 4 parts. Each point is a circuit degradation corresponding to its algorithm. Each boxplot presents the logarithmic execution times in milliseconds for all circuits. The total DKFM execution time equals the sum of the execution time of the min-cut algorithm plus the DKFM execution time. For this topology, KaHyPar+DKFM and hMetis+DKFM seems to be better than other for critical path degradation. However, we point out that PaToH produced less maximal degradation than the others. khMetis+DKFM is the third best algorithm here. Hence, we show limitations on DKFM improvement for PaToH. Note that DKFM's execution time is longer, but remember that a time limit for DKFM is set at 400 seconds, which is still reasonable for the circuit sizes processed and the results delivered.

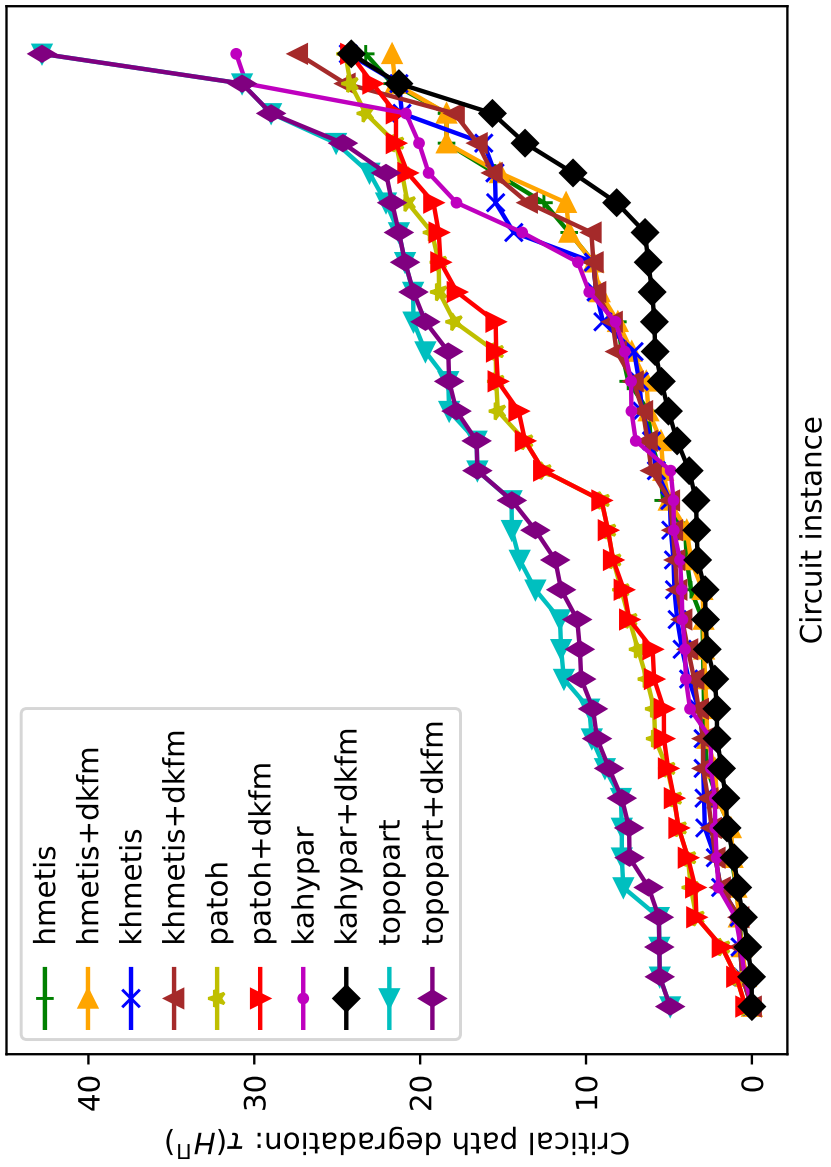


Figure 6.7: Results of improvement produced by DKFM applied to partitions computed by hMetis, khMetis, PaToH, KaHyPar, and TopoPart onto T4. For this topology, KaHyPar+DKFM seems to be better than the others except from index 28 where khMetis produced less degradation. TopoPart produced higher degradation and, DKFM shows difficulties to improve their partitions.

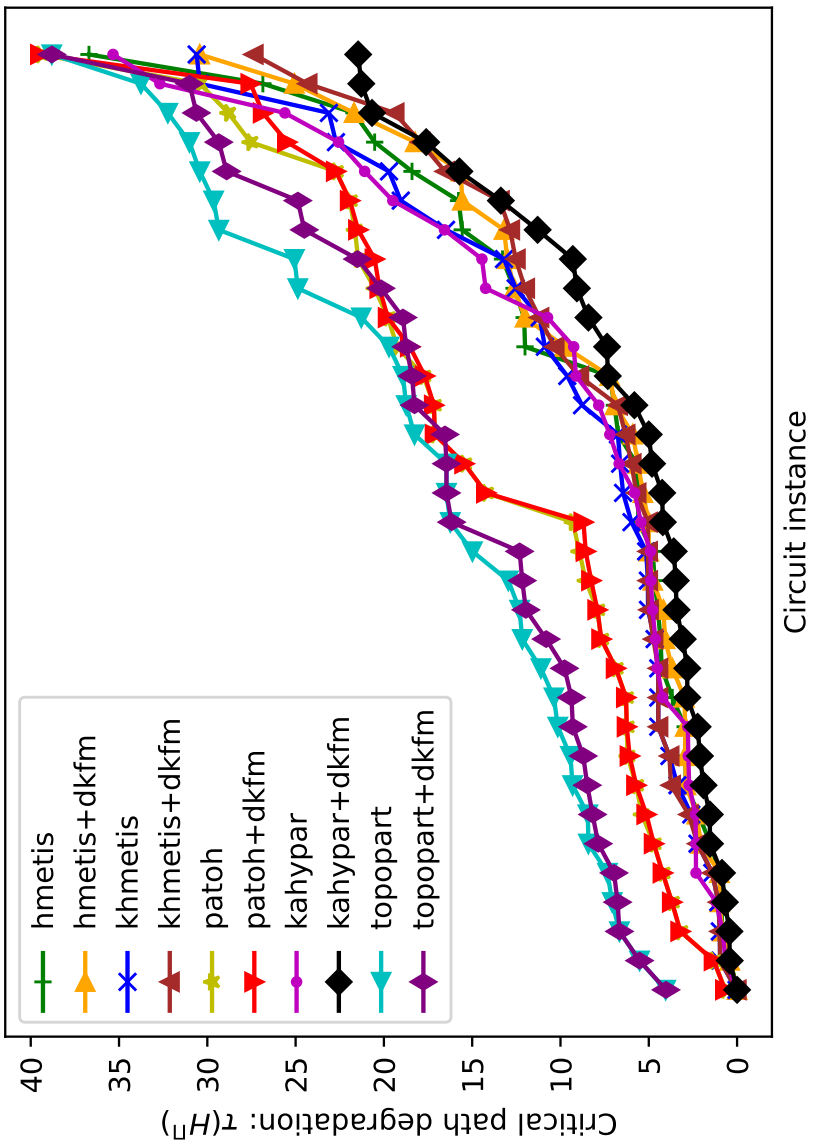


Figure 6.8: Results of improvement produced by DKFM applied to partitions computed by hMetis, khMetis, PaToH, KaHyPar, and TopoPart onto T5. For this topology, KaHyPar+DKFM seems to be better than the others. For TopoPart and PaToH, DKFM evidences difficulty at reducing routing costs sufficiently to bring degradation down to the same level as other tools' partitions.

6.2.3 Results of DKFMFAST on critical path degradation

The DKFMFAST algorithm presented as Algorithm 10 is a version of DKFM in which fewer vertices are moved and the number of cost function evaluations is reduced. Note that the cost function has $O(n)$ time complexity because the entire hypergraph must be evaluated. As a result, updating the critical path after a move or when calculating gains can be very expensive. In this subsection, we evaluate the practicality and quality of DKFMFAST compared to DKFM.

Figure 6.10 and Figure 6.11 present the results of our DKFMFAST refinement algorithm and show that the algorithm succeeds in refining partitions first computed by a min-cut algorithm. In contrast to DKFM, the DKFMFAST algorithm experiences more difficulty improving partitions. In fact, the algorithm was defined to be less accurate and faster. However, in certain cases, such as the results of KaHyPar+DKFMFAST on T1, DKFMFAST produces improvements, for example for `OneCore` and `Pulsar`, which are large circuits. Figure 6.12 presents results on fully target topology composed by 4 parts.

Figure 6.13 and Figure 6.14 present the results of our DKFMFAST refinement algorithm and show that the algorithm succeeds in refining partitions first computed by an oriented min-cut algorithm. However, similarly to previous results, DKFMFAST shows difficulties to improve partitions produced by PaToH and TopoPart. The combination of our DKFMFAST with KaHyPar seems to be preferable to obtain smaller degradation for circuits partitions. Figure 6.15 shows execution times for each algorithm, and KaHyPar appears to take longer than the others. Indeed, KaHyPar favors the computation of high quality partitions over fast execution. Consequently, if execution time is important, other tools like khMetis can be combined with DKFMFAST. Note that PaToH has a smaller execution time than other min-cut algorithms, and PaToH+DKFMFAST can produce good results with a maximum degradation close to that of hMetis+DKFMFAST and khMetis+DKFMFAST, as illustrated in Figure 6.15.

All the results associated with the figures in this section can be consulted in Appendix A.3.

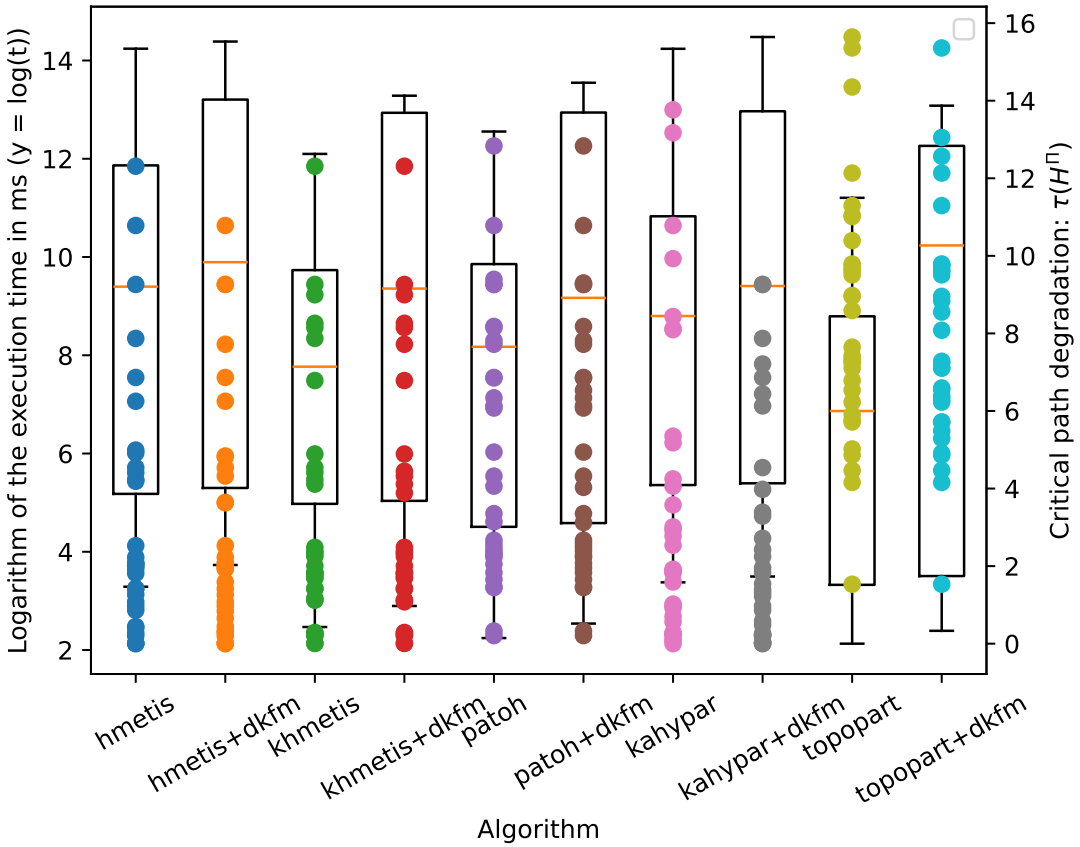


Figure 6.9: Results of improvement produced by DKFM applied to partitions computed by hMetis, khMetis, PaToH, KaHyPar, and TopoPart onto fully connected T6 composed of 8 parts. Each point is a circuit degradation corresponding to its algorithm. Each boxplot presents the logarithm of the execution times in milliseconds for all circuits. The total DKFM execution time equals the sum of the execution time of the min-cut algorithm plus the DKFM execution time. For this topology, KaHyPar+DKFM seems to be better than the others for critical path degradation. hMetis+DKFM is the second best algorithm here. Hence, we evidence limitations on DKFM improvement for PaToH.

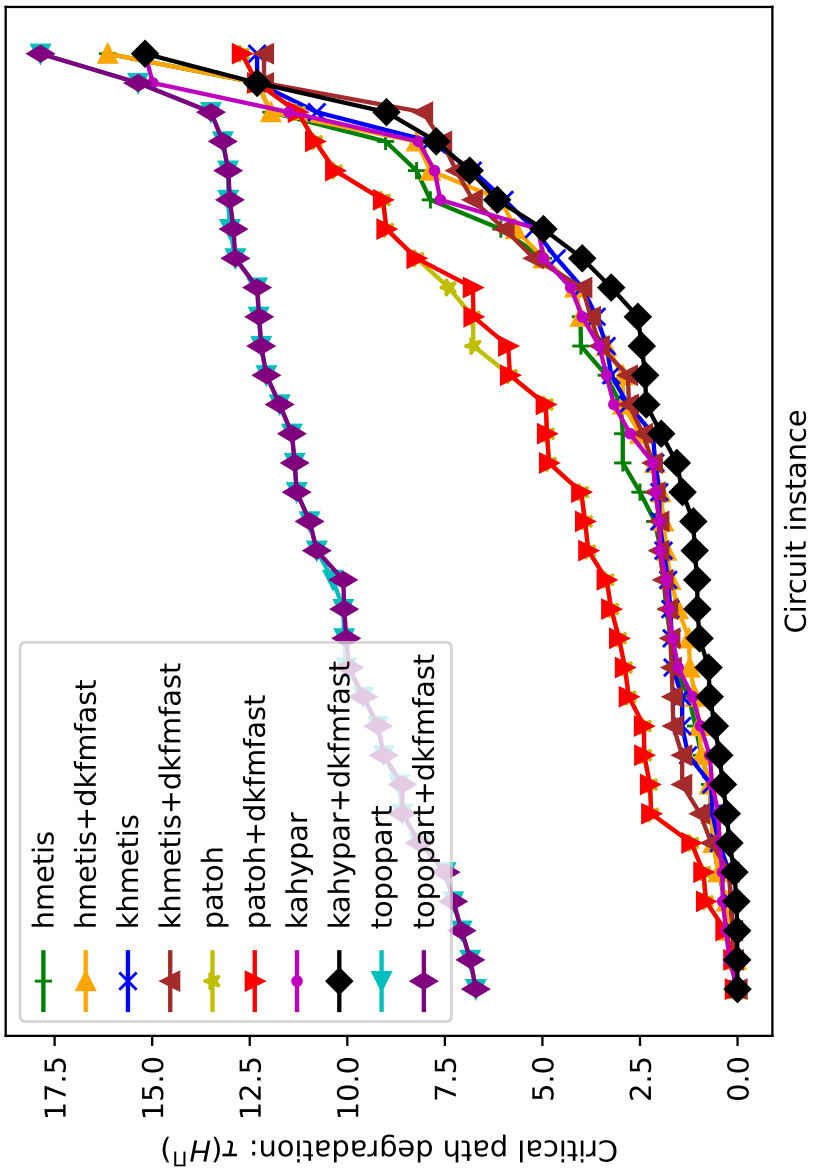


Figure 6.10: Results of improvement produced by DKFMFAST applied to partitions computed by hMetis, khMetis, PaToH, KaHyPar, TopoPart onto T1. For this topology, KaHyPar+DKFMFAST seems to be better than the others for 27 circuits. However, results of khMetis+DKFMFAST are under KaHyPar+DKFMFAST results for two instances, that is, khMetis+DKFMFAST produced less higher degradation than KaHyPar+DKFMFAST. Note that DKFMFAST shows difficulties to improves their partitions.

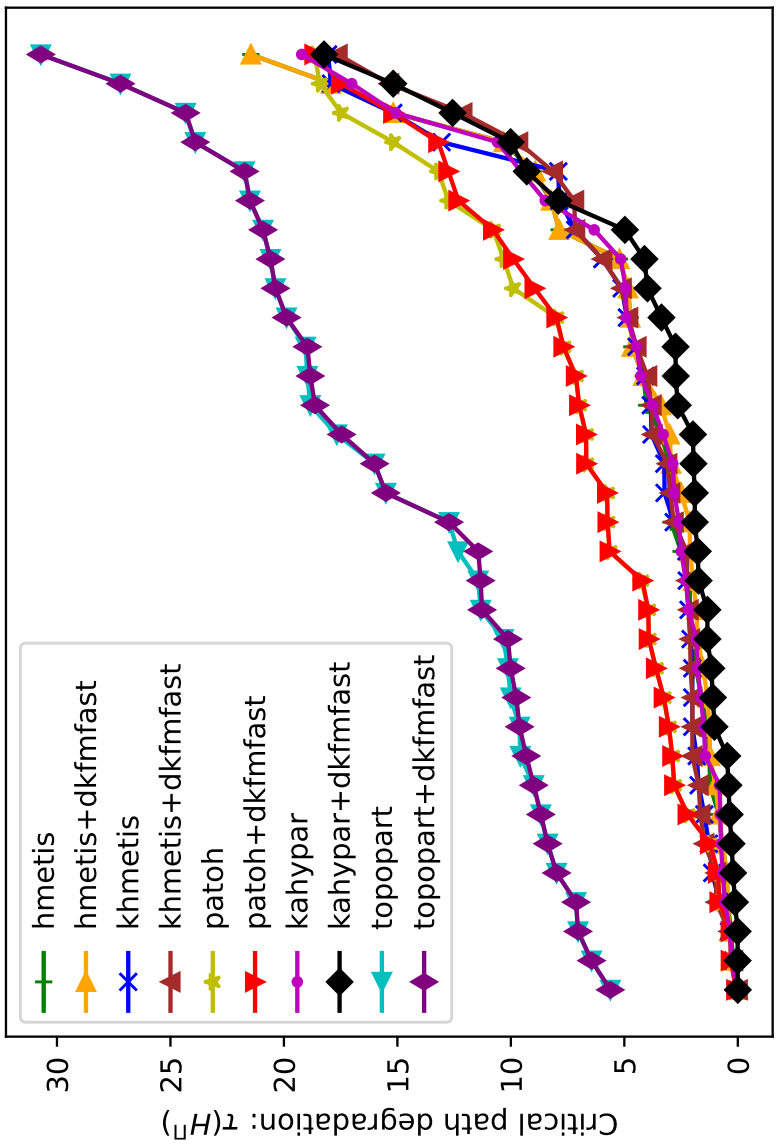


Figure 6.11: Results of improvement produced by DKFMFAST applied to partitions computed by hMetis, khMetis, PaToH, KaHyPar, and TopoPart onto T2. For this topology, KaHyPar+DKFMFAST seems to be better than the others. However, khMetis+DKFMFAST results are under KaHyPar+DKFMFAST results for two instances, that is, khMetis+DKFMFAST produced less higher degradation than KaHyPar+DKFMFAST. Note that DKFMFAST has difficulties to improves their partitions, as for T1.

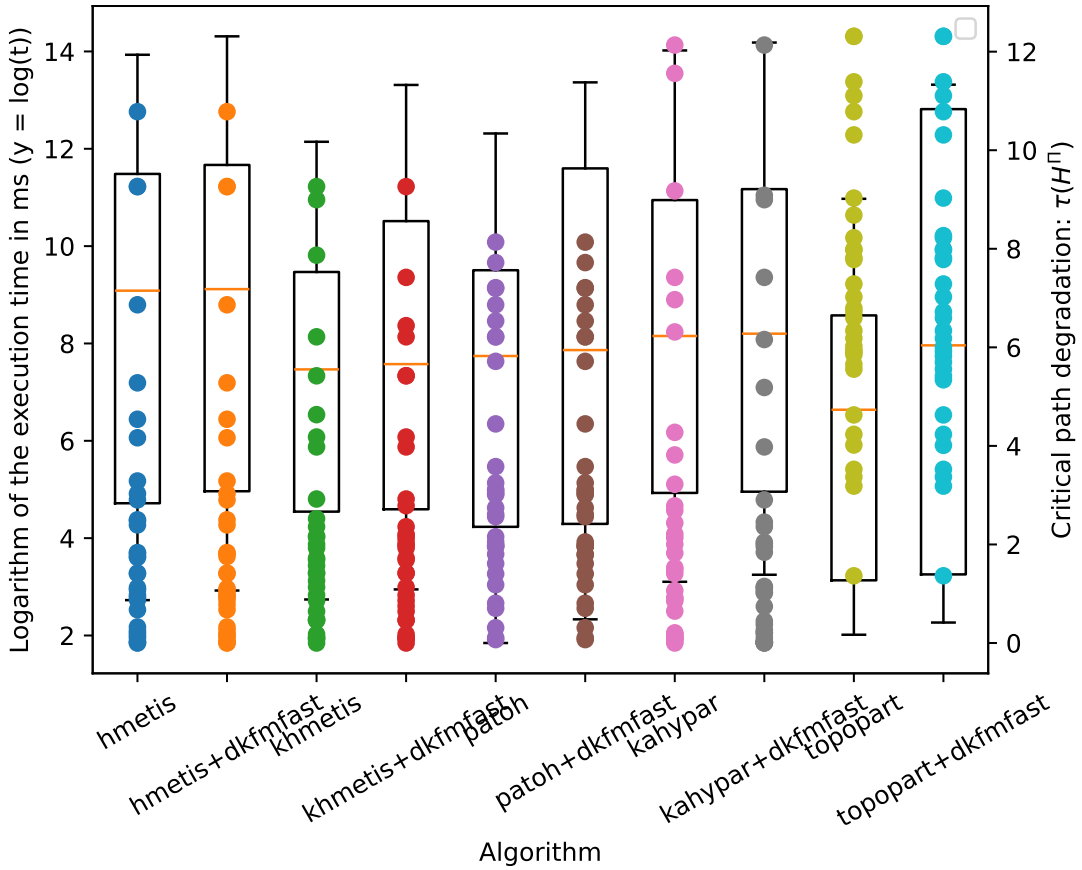


Figure 6.12: Results of improvement produced by DKFMFAST applied to partitions computed by hMetis, khMetis, PaToH, KaHyPar, and TopoPart onto fully connected T3 composed of 4 parts. Each point is a circuit degradation corresponding to its algorithm. Each boxplot presents the logarithm of the execution times in milliseconds for all circuits. The total DKFMFAST execution time equals the sum of the execution time of the min-cut algorithm plus the DKFMFAST execution time. For this topology, khMetis+DKFMFAST results on critical path degradation seems to be better than the others. We point out that PaToH produced less possible critical path degradation than the others. hMetis+DKFMFAST is the second algorithm with better results. Hence, in contrary to previous Figures, we show DKFMFAST improvement limitations for hMetis partitions. Note that the total execution times of PaToH+DKFMFAST and TopoPart +DKFMFAST, increase significantly because PaToH and TopoPart execution times are fast compared to the DKFMFAST execution time. For the other tools, DKFMFAST slightly increases the execution time compared to the total procedure.

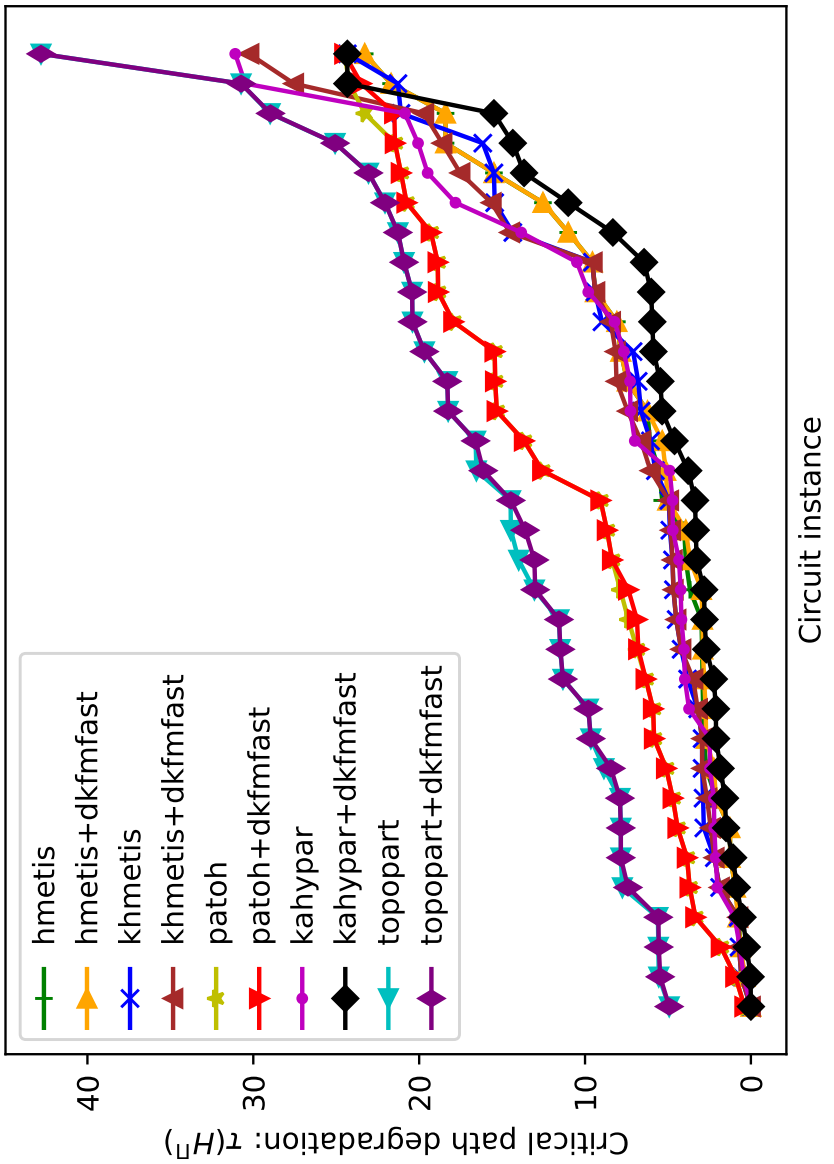


Figure 6.13: Results of improvement produced by DKFMFAST applied to partitions computed by hMetis, khMetis, PaToH, KaHyPar, TopoPart onto T4. For this topology, KaHyPar+DKFMFAST seems to produced better results than the others. However, khMetis+DKFMFAST have its degradations under degradations produced by KaHyPar+DKFMFAST, for two instances. In addition, khMetis+DKFMFAST have less higher degradation than KaHyPar+DKFMFAST. Note that DKFMFAST shows difficulties to improve their partitions.

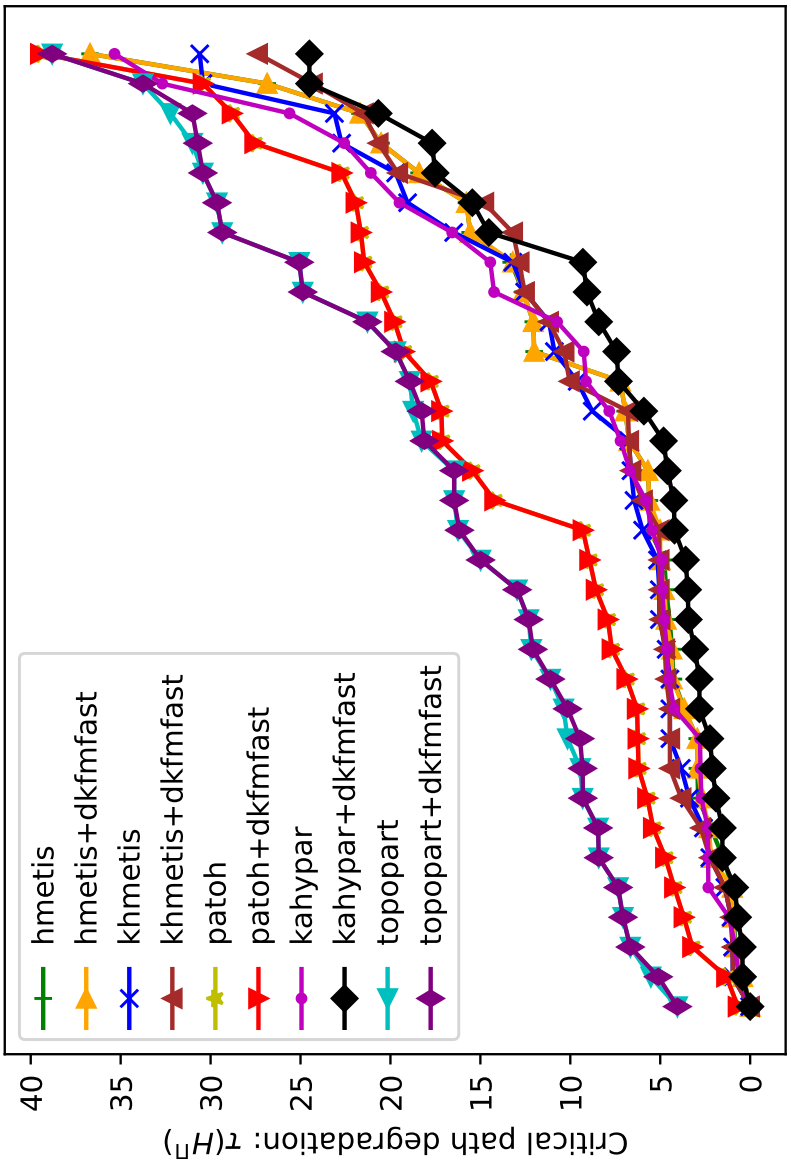


Figure 6.14: Results of improvement produced by DKFMFAST applied to partitions computed by hMetis, khMetis, PaToH, KaHyPar, TopoPart onto T5. For this topology, KaHyPar+DKFMFAST seems to produced better results than the others. DKFMFAST shows difficulties to improve their partitions.

6.2.4 Results of connectivity cost degradation with DKFM

In this subsection, we measure the effect of DKFM on connectivity cost for similar reasons exposed in Chapter 5. Indeed, this refinement algorithm only focuses on path cost, that is, vertex moves can degrade the cut size. However, as the hyperarc weight is driven by criticality, DKFM could improve connectivity cost for some partitions.

Furthermore, topology structure can have different effects on critical path degradation. Hence, each target topology induce implies a different movement strategy for DKFM. However, we noticed that results on connectivity cost evidence little variation on each topology. In addition, DKFMFAST produced similar results on connectivity cost as DKFM. For these reasons, we present the connectivity cost results for target topologies T3 in Figure 6.16 and T6 in Figure 6.16 of DKFM in the following figures. To illustrate results on each topology with DKFM and DKFMFAST, the interested reader can consult the figures in Appendix A.3. To compare connectivity cost of each partition Π, we compute a relative cost as in Equation 5.5 defined in Chapter 5.

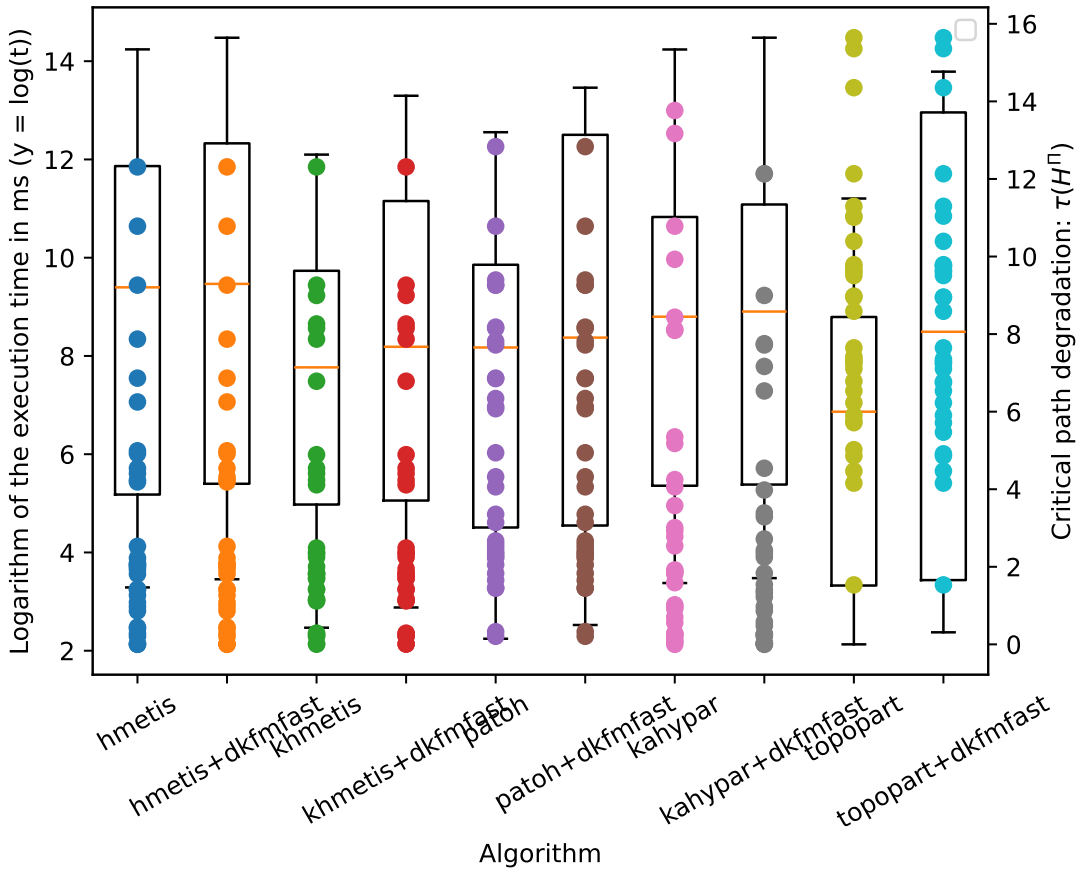


Figure 6.15: Results of improvement produced by DKFMFAST applied to partitions computed by hMetis, khMetis, PaToH, KaHyPar, and TopoPart onto fully connected T6 composed of 8 parts. Each point is a circuit degradation corresponding to its algorithm. Each boxplot presents the logarithm of the execution times in milliseconds for all circuits. The total DKFMFAST execution time equals the sum of the execution time of the min-cut algorithm plus the DKFMFAST execution time. For this topology, KaHyPar+DKFMFAST seems to produced better critical path degradation results than the others. khMetis+DKFMFAST produced the second best results here. Hence, we show DKFMFAST improvement except for hMetis, PaToH and TopoPart. The analysis of execution times is similar to previous figures.

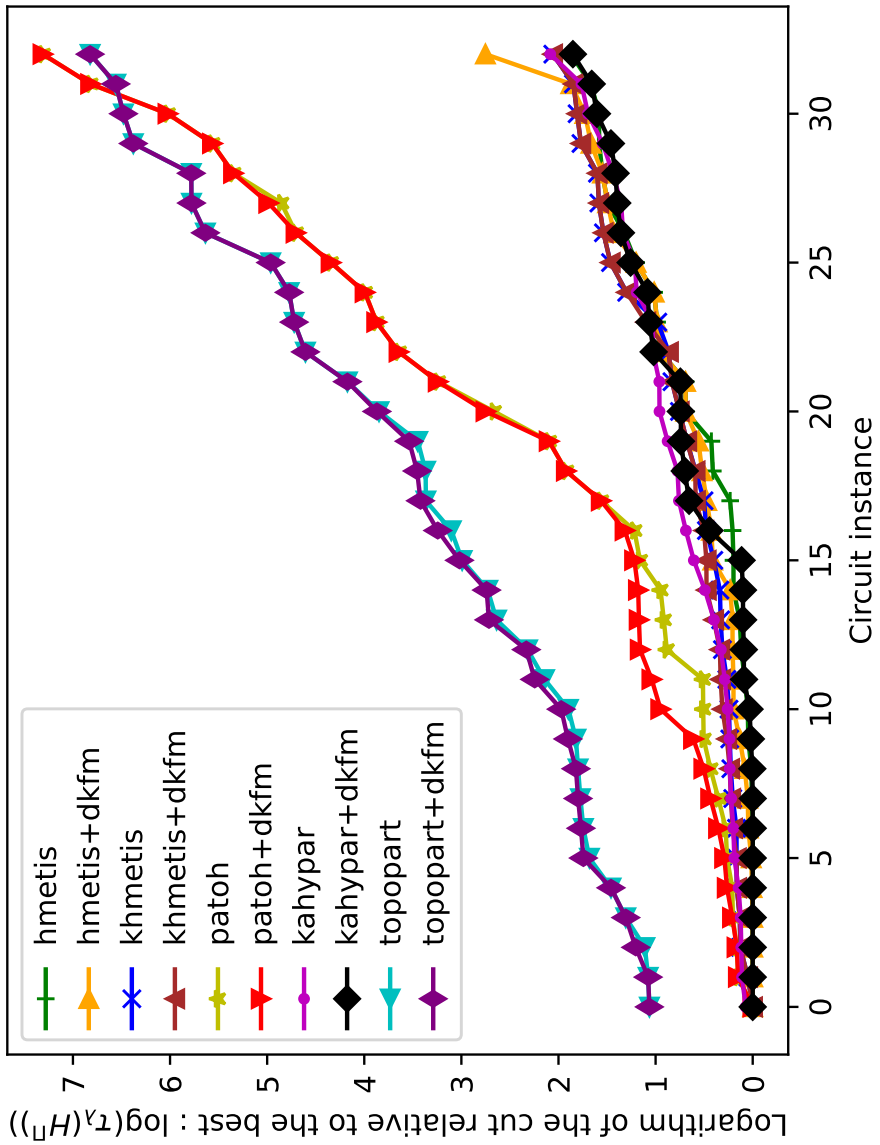


Figure 6.16: Logarithmic connectivity-minus-one cost relative to the best on target topology T3 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFM. We notice a little improvement of some partition produced by KaHyPar. However, for hMetis, khMetis, PaToH, and TopoPart, DKFM slightly reduces connectivity costs.

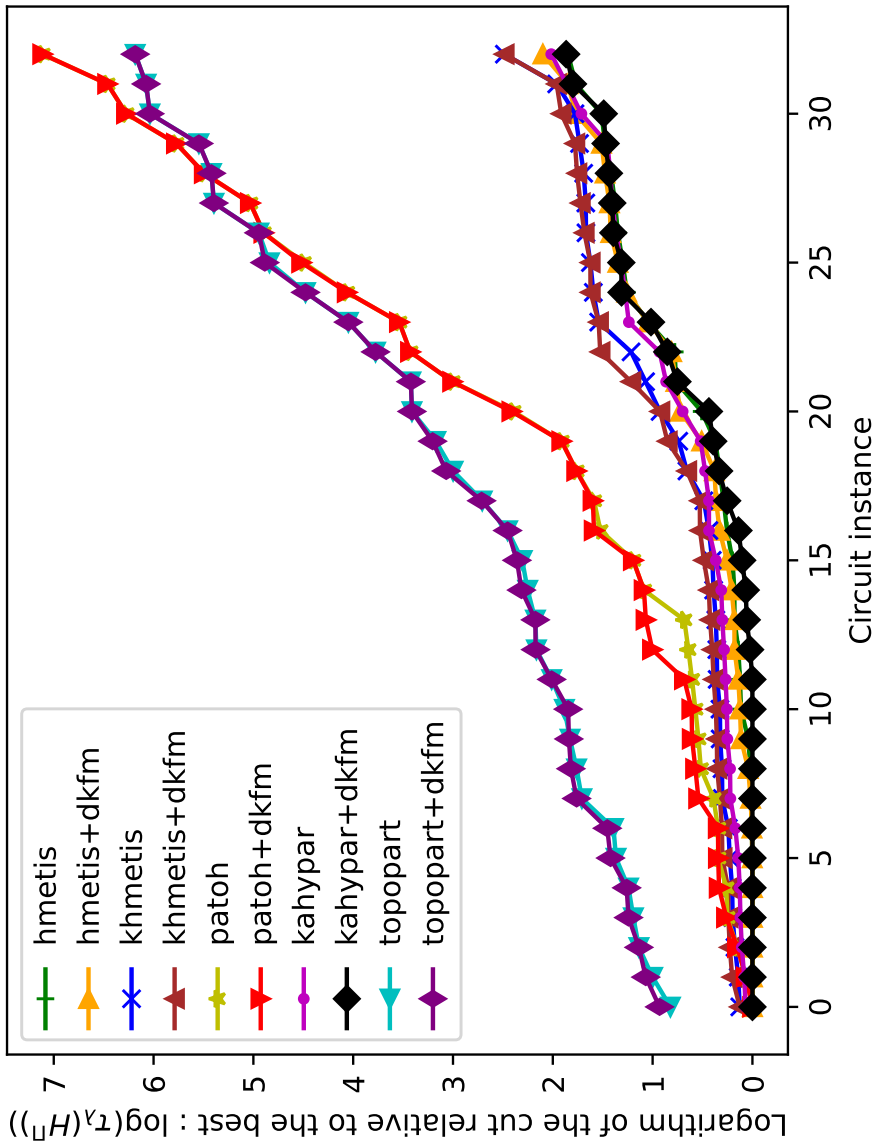


Figure 6.17: Logarithmic connectivity-minus-one cost relative to the best on target topology T6 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFM. Similar to trends on target topology T3, DKFM faintly improves some partitions produced by KaHyPar. However, for hMetis, khMetis, PaToH, and TopoPart, DKFM slightly reduces connectivity costs.

6.3 Conclusion

In this chapter, we presented an extension of the FM refinement algorithm to the problem of partitioning red-black hypergraphs, aiming at minimizing critical path degradation. This algorithm, called DKFM, considers routing costs to optimize path allocation within the partition. The calculation of the critical path when updating the gains is expensive, because all vertices must be processed. Indeed, as shown in Figure 6.2, a move can change the critical path, which can arise anywhere in the DAH. This aspect is an essential difference between the cut minimization problem and the path minimization problem. Indeed, when a vertex is moved across parts, cut costs change only for connected hyperarcs. In contrast, when minimizing f_p, moving a vertex locally has a global effect on the whole hypergraph. This property makes this problem more difficult than min-cut for FM-type local improvement algorithms.

In VLSI design, execution time is an important issue for tools. As f_p minimization involves to update the critical path after each move, DKFM is inevitably slow. Therefore, we presented DKFMFAST, a faster version of our DKFM algorithm that limits the evaluation of the critical path. DKFMFAST computes a subset of gains in addition to the cut capacity ς, which reduces the number of evaluations of the critical path during the moves.

In Section 6.2, we presented experimental results on some circuits and target topologies, both introduced in Chapter 3. The results show that, in most cases, DKFM succeeds in refining the partitions created by the min-cut tools. In this setting, DKFM only performs a maximum of 20% of the moves ($\rho = 20\%$), and the run time limit setting, is set to 400s.

We observed that DKFM experienced some trouble to improve some of the partitions derived from PaToH. Among all min-cut algorithms, the combination of KaHyPar with DKFM and DKFMFAST generally gives the best refinement outcome.

Conclusion and Perspectives

Prototyping digital electronic circuits on multi-FPGA platforms is a challenging optimization problem, which can be modeled as a constrained hypergraph partitioning problem. In this thesis, we focused on practical hypergraph partitioning with path-length degradation minimization. In this last chapter, we briefly summarize the results we obtained, and present some perspectives. We will also present a by-product of our research work, in the form of the open-source hypergraph partitioning tool RaiSin, which provides an opportunity for any researcher wishing to contribute to the red-black hypergraph partitioning problem.

Summary of the dissertation

The partitioning and mapping of digital circuits is a problem that has been studied extensively in the scientific literature over the past 40 years. After an introduction and setting the notations and definitions in Chapter 1, we presented a state of the art on circuit partitioning in Chapter 2. These research all deal with solving some form of hypergraph partitioning optimization problem. We made the point that the functions optimized by the existing algorithms do not minimize the critical path degradation during partitioning, whereas in this thesis we focused on this metric.

In addition, most of existing techniques do not take into account the target topology in which the paths are to be routed. This is due to the fact that these studies use, most often as a blackbox, a partitioning algorithm designed to minimize cut size only. Therefore, our first task, described in Chapter 1, was to model the cost function associated with our thesis problem.

We know that there is a polynomial-time algorithm for computing a critical path in a digital electronic circuit, due to wire acyclicity between two registers. However, the plain hypergraph model did not allow us to distinguish a register from a combinatorial cell. Therefore, we have defined a new enriched hypergraph model, that provides a more accurate representation of a digital electronic circuit. This model includes vertex coloring, to distinguish register cells from combinatorial ones. On this basis, we were able to define an algorithm for measuring the quality of a partition, presented in Chapter 3, as well as several algorithms for clustering, direct partitioning, and refinement, discussed in Chapters 4, 5, and 6, respectively.

Clustering algorithms reduce problem size by merging vertices of a hypergraph to partition. The clustering strategy depends on the objective function. In our case, the related clustering problem is still being studied, with results as recent as 2020. We pursued this thread by using our hypergraph model to define new clustering algorithms. Based on our hypergraph model, we improved existing

weighting schemes to identify critical vertices, *i.e.*, vertices along a critical or quasi-critical path.

The binary search clustering (BSC) Algorithm 5 presented in Chapter 4 takes profit from this weighting scheme. We compared our algorithm with the well-known Heavy Edge Matching (HEM) algorithm, and showed that our algorithm performs better for the path cost metric. In particular, we evidenced that, in order to cluster efficiently critical and quasi-critical paths, it is more interesting to perform direct clustering, as our BSC algorithm does, rather than recursive clustering, as HEM does. However, we noticed the importance of approximation biases on the paths created during the contraction phase. Indeed, when vertices are merged, false paths can be created and incorrectly considered. Consequently, maintaining consistency on combinatorial paths needs to be studied and added to contraction algorithms in order to facilitate the work of initial partitioning and refinement algorithms.

Approximation algorithms are algorithms designed to find approximate solutions to optimization problems in reasonable time. In many cases, an approximation ratio can be associated with these algorithms, which defines a provable guarantee of the distance of a solution from the optimal solution. Alongside our study of disjoint clustering, we investigated the approximation ratio between the path cost of a worst case clustering to the best one. Under some assumptions, we demonstrated that the approximation ratio can decrease from $M^2 + M$ to M, with M being the maximum cluster size.

In the context of our thesis, the number of parts should be less than or equal to the number of FPGAs of the platform. Since clustering algorithms do not prescribe the number of parts, we focused our attention to partitioning algorithms.

Due to the large size of the circuits, we first opted for greedy direct partitioning algorithms. We explored traversal algorithms such as breadth-first search and depth-first search. Subsequently, we investigated cone partitioning. Afterwards, we adapted these algorithms to favor the placement of neighbors critical vertices in the same part. Each of these adaptations is integrated into an algorithm, namely DBFS Algorithm 6, DDFS Algorithm 7, and CCP Algorithm 8. Results presented in Chapter 5 evidenced their ability to produce good solutions, especially for fully connected topologies.

Because these algorithms do not take into account target topology, the part mapping phase that take place after partitioning would not be able to reconsider vertex placement, resulting in reduced mapping efficiency. In the context of a multilevel scheme, topology-aware refinement algorithms may not be able to improve a topology-unaware initial partitioning, because a local refinement algorithm is not designed to reconsider global decisions. Consequently, target topology awareness should be integrated into these algorithms. In fact, considering the target topology

from the initial partitioning step is a main part of the perspectives of this thesis.

To broaden our spectrum of hypergraph partitioning methods, we have also experimented with an integer programming algorithm, presented in Chapter 5, and designed a model that takes into account target topologies and routing costs. Since the circuits that we consider are very large, we used our integer program only during the initial partitioning phase of a multilevel scheme.

In practice, solving the problem with integer programming is time-consuming and seems impractical. Nevertheless, obtaining solutions for some instances may allow us to compare the quality of the solutions yielded respectively by the approximation, greedy, and multilevel algorithms, with that of the exact one.

In the context of the multilevel scheme, we then logically turned our attention to refinement algorithms. In Chapter 6, we have proposed an extension of the FM algorithm, the DKFM Algorithm 11, which minimizes the critical path degradation of a partition. Because the DKFM algorithm takes into account the target topology and reduces the cost associated with routing, it significantly reduces the routing cost of a partition. For strategies using min-cut as a pre-procedure, results evidence the efficiency of DKFM as a post-procedure for minimizing routing costs.

In this thesis, we have proposed algorithms for each phase of the multilevel scheme. The combination of these algorithms creates a multilevel scheme for partitioning red-black hypergraphs. Since the multilevel scheme has proven its efficiency in partitioning hypergraphs, future algorithms aimed at minimizing path cost in partitioning red-black hypergraphs should take advantage of the multilevel scheme, combined with the proposed algorithms.

The RaiSin software

The research strategy adopted during this thesis was an empirical one: our analysis and modeling of the problem resulted in a set of ideas that were expressed as algorithms and turned into software, which allowed us to make experimental measurements of their effectiveness. All of our algorithms were implemented in the C language, within a software framework consisting of data structures to represent our red-black hypergraph model, as well as auxiliary, service routines.

The set of algorithms that we have implemented consists of HEM and BSC for coarsening, DBFS, DDFS, and CCP for partitioning, and DKFM for refinement. Each of these algorithms addresses the red-black hypergraph partitioning problem, to minimize the degradation of the critical path. This partitioning problem differs from the min-cut partitioning problem addressed by the hMetis, khMetis, KaHyPar and PaToH tools and, to our knowledge, there is no publicly avail-

able tool dedicated to our partitioning problem. Hence, we decided to formalize our work into the RaiSin software package. This tool is a by-product of this thesis, which comprises all the algorithms that we studied and developed, the various weighting schemes, routines to perform statistical analyses of hypergraphs, etc. RaiSin can be considered as the end work of this thesis, but constitutes as well a starting point to the development of algorithms for red-black hypergraph partitioning.

Appendix A

Experimental results

A.1 Numerical results of clustering algorithms

This section presents detailed results on clustering algorithms presented in Chapter 4. Each table shows results on metric $p_{\max}$, the critical path. In order to compare each algorithm, we presents results per benchmark sets introduced in Chapter 3. Then, we recall the results presented in Chapter 4. Each result is composed of a resulting critical path of each circuit. Hence, we obtain a comparison critical path obtained between heavy edge matching (HEM) algorithm 4.4.2 and binary search clustering (BSC) algorithm 5.

Table A.1: Path cost results of clustering algorithms: HEM and BSC, for circuits in ITC, Chipyard and Titan with maximum cluster size set to 2.

Instance	HEM	BSC	Instance	HEM	BSC
b01	10.46	10.16	b02	12.31	9.87
b03	10.42	9.03	b04	13.98	8.62
b05	13.89	7.28	b06	14.75	9.87
b07	8.74	7.27	b08	14.12	9.98
b09	14.35	9.96	b10	14.22	8.46
b11	7.98	7.01	b12	14.07	7.75
b13	14.06	6.66	b14	13.88	7.24
b17	13.85	7.25	b18	9.22	7.80
b19	9.35	7.97	b20	13.87	7.52
b21	9.01	7.61	b22	9.62	7.48
EightCore	10.01	7.58	mnist	14.30	10.32
mobilnet1	12.92	7.50	OneCore	13.86	7.59
PuLSAR	13.86	7.34	bitonic_mesh	12.12	9.59
cholesky_bdti	14.08	9.54	dart	9.18	6.05
denoise	13.73	7.47	des90	14.03	9.68
xge_mac	14.26	9.52			

Table A.2: Path cost results of clustering algorithms: HEM and BSC, for circuits in ITC, Chipyard and Titan with maximum cluster size set to 4.

Instance	HEM	BSC	Instance	HEM	BSC
B01	10.46	8.24	B02	12.31	9.70
B03	10.42	6.44	B04	9.61	6.96
B05	9.60	7.15	B06	12.31	9.70
B07	8.74	5.19	B08	10.81	9.03
B09	11.50	9.96	B10	9.86	8.46
B11	7.98	5.67	B12	10.56	6.64
B13	9.38	5.33	B14	9.74	5.79
B17	9.18	5.87	B18	9.22	6.94
B19	9.35	7.33	B20	9.53	6.92
B21	9.01	6.71	B22	9.62	6.28
EightCore	10.01	6.41	mnist	14.30	9.03
mobilnet1	12.92	5.01	OneCore	10.16	5.86
PuLSAR	10.40	6.47	bitonic_mesh	12.12	8.94
cholesky_bdti	11.81	9.38	dart	9.18	4.26
denoise	13.73	5.75	des90	12.77	8.42
xge_mac	11.89	8.34			

Table A.3: Path cost results of clustering algorithms: HEM and BSC, for circuits in ITC, Chipyard and Titan with maximum cluster size set to 8.

Instance	HEM	BSC	Instance	HEM	BSC
B01	10.46	8.24	B02	12.31	9.70
B03	10.42	6.25	B04	9.61	7.45
B05	9.60	6.14	B06	12.31	9.70
B07	8.74	3.66	B08	10.81	6.54
B09	11.50	9.96	B10	9.86	8.61
B11	7.98	4.93	B12	10.56	6.59
B13	9.38	2.89	B14	9.74	4.96
B17	9.18	5.13	B18	9.22	6.65
B19	9.35	6.50	B20	9.53	5.43
B21	9.01	6.29	B22	9.62	6.53
EightCore	10.01	5.68	mnist	14.30	8.75
mobilnet1	12.92	3.51	OneCore	10.16	5.53
PuLSAR	10.40	5.39	bitonic_mesh	12.12	8.29
cholesky_bdti	11.81	9.38	dart	9.18	2.87
denoise	13.73	5.19	des90	12.77	8.33
xge_mac	11.89	6.81			

Table A.4: Path cost results of clustering algorithms: HEM and BSC, for circuits in ITC, Chipyard and Titan with maximum cluster size set to 16.

Instance	HEM	BSC	Instance	HEM	BSC
B01	10.46	8.24	B02	12.31	6.90
B03	10.42	7.55	B04	9.61	6.04
B05	9.60	5.04	B06	12.31	9.70
B07	8.74	2.75	B08	10.81	4.77
B09	11.50	9.96	B10	9.86	7.21
B11	7.98	3.28	B12	10.56	5.94
B13	9.38	2.22	B14	9.74	4.00
B17	9.18	4.50	B18	9.22	6.06
B19	9.35	6.35	B20	9.53	5.00
B21	9.01	5.89	B22	9.62	5.79
EightCore	10.01	5.26	mnist	14.30	8.84
mobilnet1	12.92	3.33	OneCore	10.16	5.16
PuLSAR	10.40	5.15	bitonic_mesh	12.12	7.63
cholesky_bdti	11.81	8.59	dart	9.18	2.00
denoise	13.73	4.03	des90	12.77	7.59
xge_mac	11.89	5.89			

Table A.5: Path cost results of clustering algorithms: HEM and BSC, for circuits in ITC, Chipyard and Titan with maximum cluster size set to 32.

Instance	HEM	BSC	Instance	HEM	BSC
B01	10.31	6.17	B02	9.70	6.90
B03	9.12	7.55	B04	8.58	5.55
B05	8.01	5.36	B06	9.87	7.08
B07	7.21	2.57	B08	9.15	4.89
B09	11.50	8.54	B10	9.62	7.21
B11	6.73	3.99	B12	8.40	5.94
B13	6.61	2.12	B14	7.61	4.25
B17	7.11	3.99	B18	7.56	5.46
B19	7.83	5.74	B20	7.30	4.81
B21	7.54	5.70	B22	7.51	4.97
EightCore	8.17	4.95	mnist	14.30	8.75
mobilnet1	7.93	3.33	OneCore	7.77	4.90
PuLSAR	8.50	4.80	bitonic_mesh	12.07	7.63
cholesky_bdti	10.33	5.63	dart	6.27	1.76
denoise	11.67	3.55	des90	12.73	7.11
xge_mac	9.52	5.54			

Table A.6: Path cost results of clustering algorithms: HEM and BSC, for circuits in ITC, Chipyard and Titan with maximum cluster size set to 64.

Instance	HEM	BSC	Instance	HEM	BSC
B01	10.31	6.17	B02	7.26	0.12
B03	7.83	4.68	B04	6.69	5.07
B05	7.02	5.04	B06	9.52	0.12
B07	5.93	2.50	B08	8.20	2.46
B09	11.50	5.47	B10	8.53	5.19
B11	5.48	2.31	B12	7.50	5.99
B13	4.61	2.17	B14	6.26	4.52
B17	6.25	4.19	B18	6.70	5.89
B19	7.14	6.05	B20	6.08	4.23
B21	5.89	5.29	B22	6.31	4.66
EightCore	7.05	4.73	mnist	13.01	8.75
mobilnet1	7.43	3.33	OneCore	6.88	4.64
PuLSAR	7.52	4.45	bitonic_mesh	11.46	6.98
cholesky_bdti	9.43	4.04	dart	4.26	1.77
denoise	10.19	3.23	des90	11.51	7.03
xge_mac	8.34	5.37			

Table A.7: Path cost results of clustering algorithms: HEM and BSC, for circuits in ITC, Chipyard and Titan with maximum cluster size set to 128.

Instance	HEM	BSC	Instance	HEM	BSC
B01	10.31	0.10	B02	7.26	0.12
B03	7.83	3.66	B04	6.69	4.56
B05	7.02	4.53	B06	9.52	0.12
B07	5.93	2.07	B08	8.20	1.74
B09	11.50	2.82	B10	8.53	5.11
B11	5.48	2.08	B12	7.50	5.89
B13	4.61	2.03	B14	6.26	4.36
B17	6.25	3.54	B18	6.70	4.96
B19	7.14	6.13	B20	6.08	4.61
B21	5.89	5.52	B22	6.31	4.43
EightCore	7.05	4.59	mnist	13.01	7.55
mobilnet1	7.43	2.83	OneCore	6.88	4.34
PuLSAR	7.52	4.12	bitonic_mesh	11.46	5.94
cholesky_bdti	9.43	2.35	dart	4.26	1.50
denoise	10.19	3.07	des90	11.51	6.37
xge_mac	8.34	4.53			

Table A.8: Path cost results of clustering algorithms: HEM and BSC, for circuits in ITC, Chipyard and Titan with maximum cluster size set to 256.

Instance	HEM	BSC	Instance	HEM	BSC
B01	10.31	0.10	B02	4.64	0.12
B03	7.55	2.37	B04	6.04	2.01
B05	6.23	1.27	B06	9.52	0.12
B07	5.56	1.51	B08	8.20	0.04
B09	8.64	2.82	B10	8.53	0.05
B11	3.88	1.69	B12	6.80	3.84
B13	4.32	1.98	B14	5.56	3.74
B17	5.38	2.67	B18	5.99	4.73
B19	6.28	4.42	B20	5.35	3.79
B21	5.06	4.68	B22	5.57	4.18
EightCore	6.43	4.39	mnist	12.91	7.36
mobilnet1	5.97	2.83	OneCore	6.29	4.22
PuLSAR	6.68	3.97	bitonic_mesh	11.46	4.46
cholesky_bdti	9.38	2.35	dart	4.17	1.23
denoise	8.96	2.94	des90	11.51	4.68
xge_mac	7.16	4.28			

Table A.9: Path cost results of clustering algorithms: HEM and BSC, for circuits in ITC, Chipyard and Titan with maximum cluster size set to 512.

Instance	HEM	BSC	Instance	HEM	BSC
B01	8.24	0.10	B02	4.64	0.12
B03	7.55	0.06	B04	5.55	1.67
B05	6.14	0.62	B06	7.08	0.12
B07	5.40	1.08	B08	6.54	0.04
B09	8.64	2.82	B10	7.44	0.05
B11	3.74	2.08	B12	6.75	3.14
B13	4.17	0.03	B14	5.11	3.48
B17	4.80	2.60	B18	5.58	4.62
B19	5.66	4.06	B20	4.90	3.48
B21	4.84	4.67	B22	5.09	3.42
EightCore	6.38	3.97	mnist	12.91	6.16
mobilnet1	5.47	2.83	OneCore	5.95	3.84
PuLSAR	6.60	3.72	bitonic_mesh	11.46	2.50
cholesky_bdti	9.33	2.35	dart	3.50	0.88
denoise	8.15	2.87	des90	10.90	2.90
xge_mac	6.90	4.28			

Table A.10: Path cost results of clustering algorithms: HEM and BSC, for circuits in ITC, Chipyard and Titan with maximum cluster size set to 1024.

Instance	HEM	BSC	Instance	HEM	BSC
B01	8.24	0.10	B02	4.64	0.12
B03	7.55	0.06	B04	5.55	1.19
B05	6.14	0.37	B06	7.08	0.12
B07	5.40	0.02	B08	6.54	0.04
B09	8.64	0.07	B10	7.44	0.05
B11	3.74	1.00	B12	6.75	2.18
B13	4.17	0.03	B14	5.11	2.94
B17	4.80	1.98	B18	5.58	3.68
B19	5.66	3.34	B20	4.90	3.01
B21	4.84	3.88	B22	5.09	3.52
EightCore	6.38	2.99	mnist	12.91	6.16
mobilnet1	5.47	2.33	OneCore	5.95	2.81
PuLSAR	6.60	3.21	bitonic_mesh	11.46	1.89
cholesky_bdti	9.33	2.35	dart	3.50	0.66
denoise	8.15	2.85	des90	10.90	1.80
xge_mac	6.90	4.28			

Table A.11: Path cost results of clustering algorithms: HEM and BSC, for circuits in ITC, Chipyard and Titan with maximum cluster size set to 2048.

Instance	HEM	BSC	Instance	HEM	BSC
B01	8.09	0.10	B02	4.64	0.12
B03	7.55	0.06	B04	5.07	0.02
B05	5.62	0.01	B06	7.08	0.12
B07	4.53	0.02	B08	4.06	0.04
B09	8.54	0.07	B10	6.35	0.05
B11	3.17	1.00	B12	5.99	0.03
B13	2.98	0.03	B14	5.09	2.42
B17	4.64	1.95	B18	5.28	3.11
B19	5.44	2.37	B20	4.39	2.89
B21	4.25	2.86	B22	4.87	2.78
EightCore	5.90	1.62	mnist	12.91	6.16
mobilnet1	5.47	2.83	OneCore	5.62	1.27
PuLSAR	6.18	1.53	bitonic_mesh	10.86	0.50
cholesky_bdti	9.33	2.35	dart	3.05	0.88
denoise	7.62	2.80	des90	10.81	0.59
xge_mac	6.90	3.18			

Table A.12: Path cost results of clustering algorithms: HEM and BSC, for circuits in ITC, Chipyard and Titan with maximum cluster size set to 4096.

Instance	HEM	BSC	Instance	HEM	BSC
B01	3.94	0.10	B02	0.12	0.12
B03	6.25	0.06	B04	4.63	0.02
B05	4.64	0.01	B06	7.08	0.12
B07	4.21	0.02	B08	2.93	0.04
B09	5.68	0.07	B10	5.19	0.05
B11	2.77	1.00	B12	5.89	0.03
B13	2.84	0.03	B14	5.09	1.58
B17	4.42	1.20	B18	5.18	2.27
B19	5.29	3.07	B20	4.39	2.54
B21	4.00	2.28	B22	4.69	2.50
EightCore	5.47	1.19	mnist	11.71	6.16
mobilnet1	5.47	2.83	OneCore	5.25	0.53
PuLSAR	6.03	1.54	bitonic_mesh	10.86	0.54
cholesky_bdti	8.59	1.88	dart	3.05	0.64
denoise	7.17	2.78	des90	10.33	0.59
xge_mac	6.64	2.17			

A.2 Numerical results of initial partitioning algorithms

In this section we introduces all numeric results used to generate all figures presented in Chapter 5.

A.2.1 Critical path evaluation

In this subsection we presents all critical path obtained for each partition computed by algorithms compared in Chapter 5. These results have been used to compute the critical path degradation: $\tau(H^{\Pi})$.

Table A.13: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T1 for circuits in ITC set.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
b01	7.87	7.87	10.31	8.18	13.05	12.90	20.52	12.80
b02	12.31	12.31	9.09	15.18	12.31	15.36	21.10	8.79
b03	6.06	4.63	7.40	7.61	13.01	8.06	9.77	9.52
b04	4.03	3.35	3.35	3.36	11.42	2.89	5.66	4.24
b05	0.93	0.64	1.18	1.15	9.07	1.27	2.89	1.83
b06	11.96	12.31	12.31	14.98	15.36	15.36	18.41	12.20
b07	2.94	3.61	3.04	3.17	7.07	2.20	3.84	2.57
b08	4.98	5.22	5.83	5.11	7.29	4.80	6.13	6.03
b09	16.14	10.78	9.00	11.50	10.78	8.48	10.78	10.37
b10	8.23	6.78	8.23	7.76	9.59	6.63	10.63	10.48
b11	3.36	2.81	4.01	3.54	7.48	2.28	4.30	4.00
b12	4.16	3.24	3.24	3.98	10.10	3.76	6.59	6.75
b13	1.71	1.27	2.89	2.00	9.21	3.38	5.39	3.48
b14	2.12	2.11	2.39	2.17	11.35	1.15	3.93	2.29
b17	0.68	0.48	0.81	0.65	13.49	0.87	3.28	0.94
b18	0.10	0.21	0.11	0.21	8.15	0.52	0.41	0.32
b19	0.30	0.18	0.31	0.29	8.59	0.41	0.39	1.32
b20	4.01	2.17	2.24	1.98	11.73	1.03	3.73	2.31
b21	2.99	2.00	2.21	1.76	10.03	1.01	3.63	1.24
b22	1.63	1.78	0.88	1.52	12.27	1.13	3.99	1.80

Table A.14: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T2 for circuits in ITC set.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
b01	10.31	13.05	12.75	10.59	20.37	12.90	20.52	15.49
b02	15.18	18.05	18.41	17.01	24.33	18.23	27.20	17.94
b03	7.87	7.87	7.68	9.56	12.72	9.68	13.01	14.56
b04	4.84	4.07	5.76	4.46	9.61	3.63	8.06	8.73
b05	1.22	1.27	1.22	1.50	5.62	1.90	2.89	6.08
b06	21.46	15.18	15.18	19.22	27.20	18.23	24.15	18.29
b07	3.00	3.83	3.04	3.75	12.33	3.29	6.02	4.21
b08	5.22	7.17	6.69	6.33	15.51	6.25	11.80	8.10
b09	17.93	17.93	10.78	15.07	21.50	10.78	14.35	15.86
b10	8.23	7.91	10.32	8.49	16.01	8.15	13.44	10.48
b11	4.74	4.89	6.99	4.91	7.99	2.98	6.23	4.99
b12	4.16	3.24	3.61	4.94	11.44	4.69	10.00	9.33
b13	3.38	1.81	2.89	2.17	8.38	4.41	6.23	5.15
b14	2.12	3.23	3.94	2.80	10.13	1.70	5.05	2.29
b17	0.68	0.81	0.81	0.67	10.32	1.12	3.65	1.84
b18	0.20	0.21	0.31	0.29	6.44	0.58	0.62	0.74
b19	0.30	0.38	0.31	0.34	7.04	0.61	0.58	2.96
b20	4.01	2.17	3.26	2.29	11.33	1.54	4.59	2.31
b21	2.99	2.00	2.21	2.59	9.59	1.52	4.64	1.60
b22	1.63	3.80	0.88	1.78	8.99	1.52	4.57	1.80

Table A.15: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T3 for circuits in ITC set.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
b01	5.28	7.87	7.72	7.42	10.31	10.31	10.31	7.72
b02	9.26	9.26	6.21	12.13	12.31	9.26	12.13	5.74
b03	4.54	4.63	4.45	6.97	11.39	6.44	8.15	4.85
b04	3.03	2.15	3.03	2.69	4.23	2.43	5.66	2.72
b05	0.93	0.64	1.18	1.06	5.91	0.96	2.89	1.67
b06	9.26	6.21	6.21	11.56	12.31	12.13	12.31	9.15
b07	2.40	2.52	2.74	2.44	5.85	2.20	3.30	1.48
b08	2.90	4.18	3.58	3.82	7.29	4.18	6.07	5.00
b09	10.78	9.00	7.21	9.17	10.78	7.21	10.47	5.42
b10	6.86	5.42	6.86	6.31	7.99	5.42	6.63	5.03
b11	1.76	1.93	3.01	2.21	3.18	1.98	3.36	2.51
b12	3.29	2.36	2.16	3.22	6.80	2.66	6.49	4.94
b13	0.88	1.27	2.05	2.00	3.37	2.00	3.72	1.81
b14	1.84	1.57	1.80	1.49	4.63	1.03	3.50	1.34
b17	0.68	0.48	0.81	0.65	5.56	0.75	3.09	0.57
b18	0.10	0.10	0.11	0.12	3.52	0.41	0.21	0.32
b19	0.20	0.18	0.31	0.21	4.02	0.41	0.21	1.22
b20	2.48	2.17	1.99	1.41	6.02	0.77	3.48	1.29
b21	1.74	2.00	1.96	1.53	5.65	0.93	3.63	0.73
b22	1.12	0.86	0.70	0.85	5.93	1.01	3.73	1.68

Table A.16: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T4 for circuits in ITC set.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
b01	15.49	15.49	17.93	20.86	30.74	25.70	28.60	12.65
b02	18.41	21.28	24.51	31.09	21.28	42.62	27.38	17.76
b03	12.54	21.10	21.10	17.81	20.91	19.39	19.29	12.26
b04	7.84	8.98	7.39	7.28	9.65	6.50	11.07	10.47
b05	3.67	2.88	3.69	4.23	5.58	3.45	7.46	5.89
b06	18.41	24.33	21.46	30.48	42.80	27.55	33.48	27.26
b07	7.46	7.10	7.87	7.66	11.33	6.85	12.43	5.29
b08	8.08	15.45	15.45	13.85	14.47	13.50	14.29	11.27
b09	23.28	14.35	23.28	20.07	25.07	21.19	33.69	12.57
b10	21.69	16.16	13.68	19.49	23.05	16.16	21.69	16.01
b11	6.23	4.89	4.42	7.00	7.84	6.84	12.79	6.70
b12	9.38	6.80	15.42	8.22	14.49	9.43	16.09	7.47
b13	1.27	2.84	6.87	7.26	7.89	8.44	11.83	5.84
b14	5.53	5.76	5.84	4.90	11.57	3.63	10.39	2.63
b17	2.70	3.81	3.85	2.47	7.75	2.36	5.01	1.54
b18	0.41	0.71	0.93	0.43	4.93	1.25	1.59	0.89
b19	0.75	0.58	0.38	0.66	5.61	1.35	1.73	1.23
b20	2.88	6.05	5.11	4.72	9.80	2.30	9.51	2.77
b21	6.10	4.50	4.71	4.33	8.86	2.51	8.14	4.02
b22	4.02	4.73	3.34	3.73	7.84	2.66	9.69	5.91

Table A.17: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T5 for circuits in ITC set.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
b01	20.52	23.11	20.52	25.61	38.81	28.45	26.01	20.52
b02	18.41	30.60	27.55	32.68	30.43	39.57	24.33	15.07
b03	15.77	22.72	22.72	16.57	32.24	25.58	32.24	12.26
b04	12.06	9.61	9.30	9.13	9.44	9.41	20.94	12.95
b05	4.34	3.38	3.69	4.90	8.42	5.16	11.04	9.08
b06	36.70	30.43	21.46	35.34	21.28	33.12	30.43	27.09
b07	6.94	10.88	8.93	9.25	14.99	10.12	14.06	5.77
b08	15.57	16.48	15.45	19.50	19.71	17.64	18.43	11.15
b09	26.85	19.71	30.43	22.57	25.07	26.85	37.26	14.35
b10	21.69	19.05	17.76	21.09	16.24	17.21	28.50	13.36
b11	6.73	8.78	6.20	7.83	12.95	8.45	13.29	7.16
b12	12.01	11.18	17.12	10.73	10.15	12.89	18.98	11.86
b13	2.93	5.15	6.87	7.21	6.66	7.51	10.99	9.27
b14	5.69	6.45	7.69	6.68	9.31	4.95	13.04	4.33
b17	1.51	2.60	4.21	2.79	7.07	3.52	8.01	2.66
b18	0.38	0.98	1.32	0.66	5.52	1.77	2.37	1.10
b19	0.89	0.96	0.68	0.81	4.04	1.63	1.93	2.15
b20	5.59	5.98	6.25	5.45	8.45	3.07	10.21	3.70
b21	4.27	4.46	5.71	4.63	11.11	3.48	9.91	3.98
b22	4.49	5.05	4.72	4.79	7.34	3.34	10.59	4.43

Table A.18: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T6 for circuits in ITC set.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
b01	7.87	7.87	7.72	9.93	15.64	12.90	10.46	7.57
b02	9.26	12.31	9.26	13.17	15.36	15.18	15.18	5.92
b03	4.92	8.15	8.15	8.10	11.29	9.39	11.20	2.77
b04	4.23	4.23	3.35	3.58	8.98	4.10	7.67	4.06
b05	1.81	1.80	1.83	1.82	5.77	1.79	5.08	3.09
b06	12.31	9.26	9.26	13.76	12.13	15.36	12.13	8.97
b07	4.19	4.89	4.06	4.06	7.65	3.58	4.39	1.67
b08	6.25	8.26	6.13	8.12	10.39	6.25	8.14	4.94
b09	10.78	9.00	10.78	10.78	14.35	10.78	12.57	7.21
b10	6.86	6.78	6.86	8.44	9.51	6.71	10.79	5.19
b11	1.93	2.34	2.34	3.01	4.47	2.98	4.33	2.80
b12	4.99	4.11	4.94	4.25	7.42	4.99	7.57	4.01
b13	1.27	1.70	2.05	2.93	7.35	2.74	3.72	2.44
b14	2.00	2.37	3.14	2.54	6.54	1.89	4.62	1.73
b17	0.86	1.17	1.45	0.98	5.02	1.29	3.27	0.75
b18	0.20	0.21	0.33	0.26	4.16	0.72	0.37	0.42
b19	0.28	0.28	0.21	0.29	4.86	0.56	0.23	0.62
b20	2.02	2.48	2.48	1.90	6.79	1.28	4.41	1.63
b21	2.08	1.70	2.23	1.87	5.91	1.68	3.84	1.41
b22	2.08	1.83	1.45	1.59	6.23	1.41	4.05	2.24

Table A.19: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T1 for circuits in Chipyard and Titan sets.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
EightCore	0.62	1.66	3.91	0.49	13.06	1.05	1.36	0.83
mnist	9.02	5.97	10.83	4.97	17.86	8.15	6.53	6.53
mobilnet1	1.82	1.72	6.78	1.83	10.08	3.14	4.47	1.90
OneCore	1.25	0.11	2.40	0.47	10.37	0.63	1.91	2.70
PuLSAR	0.49	0.09	3.82	0.39	11.30	0.66	1.24	0.83
WasgaServer	0.03	0.67	4.92	0.38	13.19	1.04	1.64	1.16
bitonic_mesh	0.71	1.42	12.70	0.71	12.07	3.75	5.14	3.70
cholesky_bdti	1.90	1.90	4.83	2.10	10.96	4.50	5.10	3.85
dart	1.97	1.69	2.79	2.75	6.71	1.11	1.37	1.06
denoise	0.00	0.00	0.07	0.00	6.85	0.02	2.32	2.33
des90	2.94	1.42	11.27	0.95	12.21	3.83	5.54	6.30
xge_mac	2.50	3.98	6.77	4.28	8.67	6.85	7.20	5.63
cholesky_mc	1.08	2.00	4.88	1.67	12.87	4.60	3.73	3.96

Table A.20: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T2 for circuits in Chipyard and Titan sets.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
EightCore	0.84	2.08	3.92	0.80	20.92	1.26	1.76	1.34
mnist	9.02	5.97	17.49	5.16	30.71	9.77	9.77	19.10
mobilnet1	1.82	2.82	7.99	1.83	10.01	3.76	4.47	3.76
OneCore	1.25	1.72	4.25	0.74	17.67	0.80	1.91	6.06
PuLSAR	0.49	1.12	7.13	0.77	19.04	0.86	1.63	1.24
WasgaServer	0.44	2.26	5.74	0.56	23.91	1.24	2.07	1.24
bitonic_mesh	1.34	4.46	18.66	1.43	18.83	4.60	5.14	4.51
cholesky_bdti	4.67	2.00	6.68	2.49	19.89	5.42	5.58	5.10
dart	1.97	2.25	2.80	3.29	8.68	1.69	1.93	1.90
denoise	0.00	0.00	0.07	0.00	7.13	0.02	2.32	6.86
des90	2.94	1.47	13.20	1.62	18.97	4.60	6.22	7.83
xge_mac	2.50	5.12	9.90	4.28	20.59	8.25	8.67	12.86
cholesky_mc	2.11	2.00	5.65	2.87	21.73	5.37	5.53	5.53

Table A.21: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T3 for circuits in Chipyard and Titan sets.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
EightCore	0.15	0.46	1.61	0.08	8.68	0.84	1.14	0.57
mnist	4.17	2.92	6.53	2.78	11.11	6.06	4.17	6.16
mobilnet1	1.82	1.72	3.11	1.83	6.34	2.16	4.47	1.27
OneCore	0.32	0.11	1.96	0.17	6.60	0.43	1.91	2.28
PuLSAR	0.25	0.08	2.61	0.10	7.78	0.42	1.04	0.42
WasgaServer	0.01	0.47	2.56	0.08	7.82	0.83	1.44	0.46
bitonic_mesh	0.03	1.42	8.14	0.03	9.03	2.98	3.75	2.18
cholesky_bdti	0.97	0.97	3.25	0.90	6.18	3.57	3.25	2.11
dart	1.41	1.69	1.40	2.14	6.72	0.80	0.81	0.78
denoise	0.00	0.00	0.07	0.00	1.36	0.01	2.32	2.33
des90	1.42	1.42	7.20	0.18	8.22	3.07	4.02	3.26
xge_mac	2.50	3.98	5.72	4.28	7.02	4.15	4.24	3.98
cholesky_mc	1.08	1.13	3.09	0.92	7.98	3.57	2.33	2.11

Table A.22: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T4 for circuits in Chipyard and Titan sets.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
EightCore	0.62	3.21	8.36	2.02	16.55	2.29	2.91	2.17
mnist	11.01	9.39	19.29	9.81	22.06	15.77	13.87	12.44
mobilnet1	5.35	4.95	12.57	4.19	13.05	8.21	5.72	6.64
OneCore	1.25	1.88	6.34	2.12	14.00	2.20	2.63	4.69
PuLSAR	2.92	2.20	8.68	2.23	28.98	1.95	6.54	1.51
WasgaServer	0.93	0.77	9.06	0.98	20.40	2.31	7.62	1.85
bitonic_mesh	2.10	2.94	24.21	2.21	16.60	7.83	9.98	8.37
cholesky_bdti	2.98	4.67	20.76	4.03	18.25	12.60	14.29	7.66
dart	3.93	4.21	5.89	4.71	11.50	3.16	3.30	2.80
denoise	0.00	0.02	1.80	0.00	5.55	0.05	3.24	2.02
des90	2.94	2.94	18.88	2.47	19.68	7.69	11.41	8.98
xge_mac	9.55	9.55	18.86	10.49	20.42	15.64	17.04	12.68
cholesky_mc	2.82	6.62	15.32	3.98	18.30	12.55	6.73	7.27

Table A.23: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T5 for circuits in Chipyard and Titan sets.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
EightCore	0.99	3.83	8.57	2.33	29.35	3.30	5.82	2.17
mnist	12.63	12.54	28.81	14.24	33.76	17.86	13.87	13.01
mobilnet1	7.22	6.75	21.93	4.88	18.74	10.01	11.33	6.64
OneCore	2.18	2.24	5.39	2.76	10.37	3.01	5.81	6.81
PuLSAR	2.89	1.40	7.91	2.71	18.27	3.18	7.07	2.72
WasgaServer	0.84	1.06	14.21	1.18	24.87	3.59	7.62	2.20
bitonic_mesh	2.94	4.46	39.47	2.35	31.01	9.22	9.98	21.79
cholesky_bdti	4.67	4.67	19.78	4.22	16.46	15.37	14.29	6.73
dart	5.60	5.04	6.29	5.80	12.31	4.23	3.89	3.07
denoise	0.00	0.03	3.21	0.00	12.17	0.05	10.75	2.02
des90	3.67	4.46	19.24	2.34	18.92	9.22	11.41	10.51
xge_mac	13.28	13.20	21.64	14.45	29.64	20.17	21.64	14.33
cholesky_mc	4.78	6.62	17.17	4.52	16.46	15.21	11.35	9.12

Table A.24: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T6 for circuits in Chipyard and Titan sets.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
EightCore	0.01	1.12	2.67	0.20	9.74	1.05	2.52	0.71
mnist	4.54	4.54	9.39	5.35	11.01	6.53	4.73	5.97
mobilnet1	2.23	1.83	6.34	1.92	5.72	3.15	5.09	1.28
OneCore	0.41	0.29	2.39	0.70	11.04	0.83	2.04	2.19
PuLSAR	0.44	0.21	2.58	0.56	9.63	0.71	1.72	0.87
WasgaServer	0.01	0.01	4.32	0.08	7.24	1.14	1.44	0.52
bitonic_mesh	1.42	1.42	12.83	0.09	9.79	4.51	3.89	6.57
cholesky_bdti	0.97	2.00	7.82	0.92	8.96	5.21	4.28	2.11
dart	2.53	2.25	2.28	2.78	7.28	1.15	1.08	1.08
denoise	0.00	0.01	1.65	0.00	1.53	0.02	3.24	0.98
des90	1.42	1.42	8.18	0.24	9.61	3.89	4.02	2.90
xge_mac	4.41	4.41	6.85	5.18	8.59	6.85	6.94	3.98
cholesky_mc	1.08	1.13	6.08	1.02	7.11	4.99	3.96	5.42

A.2.2 Connectivity cost results

Results on connectivity cost presented in Chapter 5 are based on tables introduced in this subsection. Each table shows us connectivity cost of partition produced by min-cut algorithms and greedy direct partitioning algorithms DBFS, DDFS, and CCP. These results have been used to make the figures presenting the relative connectivity cost of each partition in Chapter 5.

Table A.25: Results for connectivity cost: $f_\lambda(H^\Pi)$, on target T3 for circuits in ITC set.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
b01	15778	19556	20038	14154	24038	35816	31260	23778
b02	14305	17220	17915	12819	18745	24490	23745	18830
b03	18535	29961	22100	18085	40819	89198	99463	50516
b04	43169	43144	50322	44097	120418	220814	213636	161626
b05	33429	37964	39377	41717	206369	189284	233129	166778
b06	18915	23745	26575	18244	25965	39980	37675	31050
b07	40739	50279	52293	45156	121845	159148	211547	142385
b08	29974	35699	36397	29893	55511	97342	112900	66342
b09	24323	23430	29826	24716	53256	95654	122330	72795
b10	35580	37898	46031	32682	76662	108833	115062	73524
b11	45562	58161	60175	52722	132001	149135	251713	143654
b12	40711	42731	44628	40933	194358	282611	322895	225156
b13	2506	5757	6259	6996	55841	83411	98252	55752
b14	228525	309836	375903	293802	2303830	2383284	2803858	1334600
b17	153008	187882	236399	227893	4440854	4756984	2957179	3059983
b18	100631	141055	143351	241692	6535172	7469478	3479094	3836982
b19	375407	286119	145289	380083	14537494	14686802	5477027	5932392
b20	313765	300327	425778	339824	3565166	2929169	5148867	2541813
b21	296617	297894	391461	284082	3598381	2813471	4855528	2194020
b22	365413	354106	348225	366180	5974029	3825984	7534837	3210974

Table A.26: Results for connectivity cost: $f_\lambda(H^\Pi)$, on target T6 for circuits in ITC set.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
b01	25520	34260	34038	25930	45778	46816	51260	39778
b02	24220	27135	25830	21177	33575	29490	30490	26745
b03	34584	52041	45012	33862	73376	114858	129589	69181
b04	74754	91111	104947	79659	245079	289822	347826	235212
b05	63773	73487	74196	75662	250078	267831	414111	278945
b06	30880	39880	41100	29310	41065	54250	52725	42795
b07	65307	89406	92451	67886	142333	213239	286311	168333
b08	51394	64639	61169	47776	83960	123696	146367	92934
b09	44575	52865	51758	39742	86544	141335	159011	84510
b10	62665	72615	85337	57906	99565	145142	171088	110970
b11	73524	99794	92817	81991	225165	213933	363258	201455
b12	81812	86006	93818	81460	290533	404108	492064	278364
b13	9186	14946	17448	15426	87312	123897	138051	70520
b14	437975	463584	637360	513791	2363410	3014789	4120905	1681932
b17	457662	563983	538762	522558	5729510	6109965	6482122	4441082
b18	273978	390123	292301	395372	8302637	9814365	7187942	4877008
b19	523605	479477	376702	605313	16437469	17746733	9192039	6504057
b20	630083	653956	811173	607296	4740326	4672276	7772277	3204095
b21	643990	626356	884246	624565	4347590	5030079	7412459	2831939
b22	779010	803104	983400	693716	6241214	5503767	11355775	4242770

Table A.27: Results for connectivity cost: $f_\lambda(H^\Pi)$, on target T3 for circuits in Chipyard and Titan sets.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
EightCore	339366	546691	16334625	471182	40149338	56014398	46144242	29082489
mnist	59130	76928	1558202	47077	3929594	3777581	2824908	1164321
mobilnet1	124962	134810	21324917	174563	57561026	70961807	49586307	11050178
OneCore	291397	354531	3662963	284467	8019800	8871216	8607543	5199406
PuLSAR	550196	673694	24496978	738647	50737172	74042746	54228692	22483570
WasgaServer	433534	565709	91152841	550997	121510727	283920687	197110509	80590091
bitonic_mesh	211714	231620	71051106	180632	43656449	90744621	58383626	34236135
cholesky_bdti	309297	289504	24165866	318991	34461861	120710092	65936089	10720266
dart	231192	220916	8449843	279801	10362245	28311031	31022222	14749597
denoise	8623	9078	611919	63909	2576592	1465727	5957056	4458924
des90	172754	215505	33674895	155232	23958572	48543762	34287651	15251078
xge_mac	84212	92796	647476	83697	733536	1160468	1396644	807264
cholesky_mc	170302	203212	12474933	244881	16160263	45709620	26754025	5214425

Table A.28: Results for connectivity cost: $f_\lambda(H^\Pi)$, on target T6 for circuits in Chipyard and Titan sets.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
EightCore	553487	1066992	32142627	724250	48723133	69334808	62698525	32930447
mnist	144724	203838	3370120	153079	4748696	4718962	3692758	1836766
mobilnet1	261147	353528	34894439	402837	66785102	85771714	63309427	14240326
OneCore	697722	884140	6266748	712944	13627759	11460042	11007493	7142899
PuLSAR	1506550	1813541	38721456	1670173	72032313	92282919	75022023	31249821
WasgaServer	752608	907819	163202560	773565	147895883	345113421	236595129	99840027
bitonic_mesh	441097	499447	131346603	383180	46519734	126159394	70490520	34816154
cholesky_bdti	667277	672459	60250821	731404	47576242	140662911	78759613	11646553
dart	397816	443324	13440289	434285	11958429	33654062	37622771	16666335
denoise	14113	169442	4536172	105925	3186836	1874952	28617078	2704800
des90	379460	423979	53545126	357375	40545626	66288646	41099206	20463302
xge_mac	222308	297860	1133248	220018	1125740	1928312	1854524	1045100
cholesky_mc	291529	394863	20750467	321481	19019964	52712091	31216682	6398397

A.2.3 Balance cost of partition

In Chapter 5, we presented a comparison on vertex weight balance of partition. For this purpose, we use the results presented in the following tables.

Table A.29: Results for balance cost: $\beta(H^{\Pi})$, on target T3 for circuits in ITC set.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
b01	4.92	1.00	1.00	1.00	2.96	1.00	2.96	2.96
b02	4.33	1.00	1.00	1.00	1.00	1.00	1.00	1.00
b03	1.00	1.00	1.00	1.62	2.23	1.62	1.62	2.23
b04	1.93	1.53	1.00	2.01	2.20	2.20	2.20	2.33
b05	4.30	1.00	1.19	1.79	2.26	2.16	2.16	2.26
b06	6.17	1.00	1.00	1.00	2.72	1.00	1.00	2.72
b07	6.18	1.68	1.23	2.04	2.35	2.13	2.13	2.35
b08	3.16	1.54	1.00	2.08	1.54	1.54	1.54	2.08
b09	5.65	1.58	1.00	2.04	2.74	2.16	2.16	2.74
b10	4.37	1.00	1.48	1.58	2.44	1.96	1.96	2.44
b11	5.09	1.38	1.26	1.97	2.28	2.15	2.15	2.28
b12	4.71	1.93	1.00	2.07	2.21	2.11	2.11	2.21
b13	4.54	1.82	1.00	1.51	2.09	2.09	2.09	2.36
b14	5.56	2.15	1.26	2.08	2.25	2.24	2.24	2.25
b17	4.13	2.14	1.49	2.22	2.25	2.25	2.25	2.25
b18	1.96	1.97	1.50	2.15	2.25	2.25	2.25	2.25
b19	1.98	1.16	1.26	2.19	2.13	2.25	2.25	2.25
b20	2.13	1.67	1.42	2.20	2.22	2.24	2.24	2.25
b21	1.68	1.13	1.31	1.98	2.25	2.25	2.25	2.25
b22	1.64	2.18	1.35	2.18	2.25	2.25	2.25	2.25

Table A.30: Results for balance cost: $\beta(H^{\Pi})$, on target T6 for circuits in ITC set.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
b01	4.92	1.00	1.00	1.00	1.00	1.00	1.00	1.00
b02	4.33	1.00	1.00	1.00	1.00	1.00	1.00	1.00
b03	2.23	1.00	1.00	1.62	1.62	1.62	1.00	1.62
b04	2.86	1.27	1.00	1.50	1.67	1.67	1.53	1.67
b05	3.91	1.00	1.00	1.51	1.68	1.68	1.58	1.68
b06	4.45	1.00	1.00	1.00	1.00	1.00	1.00	1.00
b07	4.60	1.23	1.00	1.45	1.68	1.68	1.45	1.68
b08	3.16	1.00	1.00	1.38	1.54	1.54	1.00	1.54
b09	3.33	1.00	1.00	1.58	1.58	1.58	1.00	1.58
b10	3.88	1.48	1.00	1.48	1.96	1.96	1.48	1.96
b11	3.55	1.38	1.00	1.47	1.64	1.64	1.51	1.64
b12	3.13	1.56	1.09	1.54	1.65	1.65	1.56	1.65
b13	4.27	1.27	1.00	1.54	1.82	1.82	1.54	1.82
b14	4.08	1.37	1.04	1.61	1.61	1.62	1.61	1.62
b17	1.51	1.50	1.17	1.62	1.63	1.63	1.62	1.63
b18	1.49	1.48	1.25	1.61	1.62	1.63	1.62	1.63
b19	1.40	1.39	1.13	1.56	1.62	1.63	1.62	1.63
b20	1.58	1.58	1.17	1.59	1.63	1.63	1.62	1.63
b21	1.58	1.51	1.11	1.60	1.62	1.63	1.62	1.63
b22	1.57	1.59	1.16	1.59	1.63	1.63	1.62	1.63

Table A.31: Results for balance cost: $\beta(H^{\Pi})$, on target T3 for circuits in Chipyard and Titan sets.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
EightCore	5.07	2.14	1.25	2.23	1.45	2.25	2.25	2.25
mnist	3.15	1.00	1.49	1.77	2.26	2.25	2.25	2.26
mobilnet1	3.66	1.24	1.25	2.19	2.25	2.25	2.25	2.25
OneCore	5.98	2.11	1.00	2.19	2.25	2.25	2.25	2.25
PuLSAR	4.91	1.84	1.30	2.13	2.25	2.25	2.25	2.25
WasgaServer	5.62	2.18	1.50	2.06	2.24	2.25	2.25	2.25
bitonic_mesh	3.38	1.00	1.25	1.74	2.25	2.25	2.25	2.25
cholesky_bdti	4.01	1.22	1.00	1.91	1.55	2.25	2.25	2.25
dart	1.81	1.00	1.25	2.21	2.25	2.25	2.25	2.25
denoise	2.95	1.73	1.25	2.17	1.55	2.25	2.25	2.25
des90	3.50	1.00	1.00	1.89	2.25	2.25	2.25	2.25
xge_mac	5.07	1.52	1.43	2.25	2.25	2.23	2.23	2.25
cholesky_mc	4.58	1.19	1.25	2.12	1.87	2.25	2.25	2.25

Table A.32: Results for balance cost: $\beta(H^{\Pi})$, on target T6 for circuits in Chipyard and Titan sets.

Instance	hMetis	khMetis	PaToH	KaHyPar	TopoPart	DBFS	DDFS	CCP
EightCore	4.48	1.57	1.08	1.61	1.61	1.62	1.62	1.63
mnist	1.96	1.30	1.24	1.61	1.63	1.62	1.62	1.63
mobilnet1	3.69	1.29	1.16	1.59	1.62	1.62	1.62	1.62
OneCore	4.87	1.56	1.16	1.61	1.62	1.62	1.62	1.62
PuLSAR	5.04	1.45	1.17	1.62	1.63	1.62	1.62	1.63
WasgaServer	4.14	1.59	1.17	1.57	1.62	1.62	1.62	1.63
bitonic_mesh	2.18	1.00	1.13	1.49	1.62	1.62	1.62	1.62
cholesky_bdti	2.62	1.31	1.08	1.55	1.60	1.62	1.62	1.63
dart	2.63	1.00	1.08	1.62	1.62	1.62	1.62	1.63
denoise	2.70	1.50	1.17	1.61	1.63	1.62	1.62	1.63
des90	3.07	1.19	1.08	1.52	1.62	1.62	1.62	1.63
xge_mac	3.16	1.30	1.16	1.61	1.61	1.59	1.59	1.61
cholesky_mc	4.02	1.26	1.08	1.49	1.61	1.62	1.62	1.63

A.3 Numerical results of refinement algorithms: DKFM and DKFMFAST

In this section we introduces all numeric results used to generate all figures presented in Chapter 6.

A.3.1 Numerical results of DKFM

Critical path results

Results on critical path degradation presented in Chapter 6 are based on tables introduced in this subsection. Each table shows us effect of DKFMFAST on critical path degradation of partition produced by min-cut algorithms.

Table A.33: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T1 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	7.87	7.87	7.87	7.27	10.31	10.31	8.18	7.72	13.05	12.90
b02	12.31	12.31	12.31	12.13	9.09	9.09	15.18	12.13	12.31	12.13
b03	6.06	4.63	4.63	7.87	7.40	6.06	7.61	5.97	13.01	13.01
b04	4.03	3.89	3.35	3.03	3.35	2.75	3.36	2.22	11.42	11.42
b05	0.93	0.93	0.64	0.96	1.18	1.18	1.15	0.74	9.07	8.76
b06	11.96	9.26	12.31	12.13	12.31	12.31	14.98	12.31	15.36	15.18
b07	2.94	2.94	3.61	2.78	3.04	3.04	3.17	2.55	7.07	7.07
b08	4.98	4.98	5.22	5.22	5.83	5.83	5.11	3.70	7.29	6.25
b09	16.14	12.57	10.78	7.21	9.00	9.00	11.50	9.00	10.78	10.58
b10	8.23	8.07	6.78	6.63	8.23	8.23	7.76	6.78	9.59	9.43
b11	3.36	3.31	2.81	2.37	4.01	3.86	3.54	2.33	7.48	7.48
b12	4.16	4.16	3.24	3.76	3.24	3.24	3.98	3.24	10.10	9.38
b13	1.71	0.88	1.27	2.10	2.89	2.89	2.00	1.71	9.21	7.54
b14	2.12	2.12	2.11	2.42	2.39	2.39	2.17	1.56	11.35	9.36
b17	0.68	0.67	0.48	1.65	0.81	0.81	0.65	0.37	13.49	13.13
b18	0.10	0.21	0.21	0.21	0.11	0.11	0.21	0.21	8.15	8.15
b19	0.30	0.28	0.18	0.18	0.31	0.31	0.29	0.28	8.59	8.59
b20	4.01	1.52	2.17	2.17	2.24	1.80	1.98	1.03	11.73	10.27
b21	2.99	0.98	2.00	2.00	2.21	2.17	1.76	1.03	10.03	9.85
b22	1.63	1.22	1.78	1.76	0.88	0.88	1.52	0.47	12.27	11.89

Table A.34: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T2 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
B01	10.31	10.31	13.05	10.31	12.75	12.60	10.59	10.01	20.37	17.93
B02	15.18	15.01	18.05	12.31	18.41	18.41	17.01	15.18	24.33	21.46
B03	7.87	7.78	7.87	9.30	7.68	7.68	9.56	8.15	12.72	12.72
B04	4.84	4.84	4.07	3.99	5.76	5.62	4.46	2.54	9.61	9.05
B05	1.22	1.22	1.27	0.96	1.22	1.22	1.50	1.17	5.62	5.62
B06	21.46	21.10	15.18	12.13	15.18	15.18	19.22	15.18	27.20	21.46
B07	3.00	3.00	3.83	2.88	3.04	3.04	3.75	2.74	12.33	11.05
B08	5.22	5.22	7.17	7.17	6.69	5.83	6.33	3.70	15.51	12.46
B09	17.93	14.35	17.93	7.21	10.78	10.78	15.07	10.78	21.50	19.71
B10	8.23	8.07	7.91	7.91	10.32	8.23	8.49	7.83	16.01	15.85
B11	4.74	3.92	4.89	4.36	6.99	5.82	4.91	3.36	7.99	7.08
B12	4.16	4.16	3.24	3.76	3.61	3.24	4.94	4.11	11.44	11.18
B13	3.38	1.81	1.81	2.10	2.89	2.89	2.17	1.95	8.38	8.38
B14	2.12	2.12	3.23	2.84	3.94	3.40	2.80	1.98	10.13	10.13
B17	0.68	0.67	0.81	1.65	0.81	0.81	0.67	0.40	10.32	7.88
B18	0.20	0.21	0.21	0.21	0.31	0.31	0.29	0.21	6.44	6.44
B19	0.30	0.38	0.38	0.38	0.31	0.31	0.34	0.28	7.04	6.84
B20	4.01	1.52	2.17	2.17	3.26	3.20	2.29	1.03	11.33	11.31
B21	2.99	0.98	2.00	2.00	2.21	2.20	2.59	1.73	9.59	9.36
B22	1.63	1.79	3.80	3.80	0.88	0.88	1.78	0.47	8.99	8.82

Table A.35: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T3 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	5.28	5.28	7.87	7.27	7.72	7.72	7.42	5.28	10.31	10.31
b02	9.26	9.26	9.26	9.26	6.21	6.21	12.13	9.26	12.31	9.26
b03	4.54	4.54	4.63	6.25	4.45	4.45	6.97	4.92	11.39	9.77
b04	3.03	3.03	2.15	2.79	3.03	2.58	2.69	1.98	4.23	4.23
b05	0.93	0.93	0.64	0.64	1.18	1.18	1.06	0.74	5.91	5.91
b06	9.26	9.09	6.21	6.21	6.21	6.21	11.56	9.09	12.31	9.26
b07	2.40	2.40	2.52	2.21	2.74	2.74	2.44	1.98	5.85	5.40
b08	2.90	2.90	4.18	4.18	3.58	3.14	3.82	2.66	7.29	7.29
b09	10.78	9.00	9.00	6.80	7.21	7.21	9.17	7.21	10.78	10.78
b10	6.86	6.71	5.42	5.26	6.86	6.86	6.31	5.11	7.99	7.99
b11	1.76	1.48	1.93	1.43	3.01	2.51	2.21	1.79	3.18	3.18
b12	3.29	3.29	2.36	2.31	2.16	2.16	3.22	2.36	6.80	6.54
b13	0.88	0.43	1.27	2.05	2.05	2.00	2.00	1.71	3.37	3.08
b14	1.84	1.82	1.57	1.99	1.80	1.80	1.49	1.15	4.63	4.50
b17	0.68	0.67	0.48	0.75	0.81	0.81	0.65	0.37	5.56	5.51
b18	0.10	0.11	0.10	0.10	0.11	0.11	0.12	0.10	3.52	3.52
b19	0.20	0.18	0.18	0.18	0.31	0.31	0.21	0.20	4.02	4.02
b20	2.48	1.11	2.17	2.17	1.99	1.96	1.41	1.03	6.02	5.77
b21	1.74	0.98	2.00	2.00	1.96	1.92	1.53	1.03	5.65	5.63
b22	1.12	0.78	0.86	0.86	0.70	0.70	0.85	0.45	5.93	5.89

Table A.36: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T4 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
B01	15.49	15.19	15.49	15.64	17.93	17.78	20.86	15.64	30.74	30.74
B02	18.41	18.41	21.28	24.51	24.51	21.46	31.09	21.28	21.28	21.28
B03	12.54	11.20	21.10	9.68	21.10	14.06	17.81	8.15	20.91	20.91
B04	7.84	7.24	8.98	7.14	7.39	7.39	7.28	5.83	9.65	9.58
B05	3.67	3.67	2.88	3.48	3.69	3.41	4.23	2.80	5.58	5.58
B06	18.41	18.41	24.33	27.38	21.46	21.46	30.48	24.15	42.80	42.80
B07	7.46	6.37	7.10	8.16	7.87	7.72	7.66	6.01	11.33	10.37
B08	8.08	8.08	15.45	16.54	15.45	15.45	13.85	6.25	14.47	14.47
B09	23.28	21.50	14.35	17.93	23.28	22.87	20.07	10.78	25.07	24.65
B10	21.69	21.69	16.16	13.52	13.68	13.68	19.49	13.68	23.05	21.69
B11	6.23	6.23	4.89	6.29	4.42	4.42	7.00	5.03	7.84	7.40
B12	9.38	9.17	6.80	8.40	15.42	15.42	8.22	6.44	14.49	11.86
B13	1.27	1.27	2.84	2.64	6.87	5.98	7.26	4.51	7.89	6.23
B14	5.53	5.48	5.76	4.76	5.84	5.30	4.90	3.78	11.57	10.53
B17	2.70	2.70	3.81	2.43	3.85	3.85	2.47	1.50	7.75	7.38
B18	0.41	0.35	0.71	0.71	0.93	0.93	0.43	0.25	4.93	4.93
B19	0.75	0.79	0.58	0.58	0.38	0.38	0.66	0.51	5.61	5.61
B20	2.88	2.73	6.05	6.05	5.11	5.05	4.72	3.33	9.80	9.34
B21	6.10	5.02	4.50	4.50	4.71	4.69	4.33	3.27	8.86	8.61
B22	4.02	2.88	4.73	4.63	3.34	3.34	3.73	2.69	7.84	7.84

Table A.37: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T5 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	20.52	15.64	23.11	17.93	20.52	20.52	25.61	20.67	38.81	38.81
b02	18.41	18.23	30.60	24.33	27.55	27.55	32.68	21.28	30.43	24.51
b03	15.77	13.01	22.72	11.20	22.72	22.72	16.57	11.29	32.24	30.62
b04	12.06	9.76	9.61	8.94	9.30	8.70	9.13	7.32	9.44	9.30
b05	4.34	4.08	3.38	4.74	3.69	3.69	4.90	3.43	8.42	7.86
b06	36.70	30.43	30.43	27.38	21.46	21.46	35.34	21.46	21.28	18.41
b07	6.94	6.91	10.88	10.34	8.93	8.26	9.25	7.36	14.99	8.48
b08	15.57	15.57	16.48	16.54	15.45	15.45	19.50	13.38	19.71	16.54
b09	26.85	25.07	19.71	19.40	30.43	26.85	22.57	15.72	25.07	21.50
b10	21.69	21.69	19.05	13.44	17.76	17.68	21.09	17.60	16.24	16.16
b11	6.73	5.91	8.78	6.32	6.20	6.08	7.83	5.82	12.95	11.96
b12	12.01	12.01	11.18	12.01	17.12	17.12	10.73	9.07	10.15	9.38
b13	2.93	2.93	5.15	5.05	6.87	6.87	7.21	5.00	6.66	6.66
b14	5.69	5.33	6.45	5.00	7.69	7.69	6.68	4.81	9.31	8.71
b17	1.51	2.29	2.60	2.74	4.21	4.21	2.79	1.53	7.07	6.77
b18	0.38	0.42	0.98	0.98	1.32	1.32	0.66	0.42	5.52	5.52
b19	0.89	1.00	0.96	0.96	0.68	0.68	0.81	0.69	4.04	4.03
b20	5.59	5.01	5.98	5.98	6.25	6.25	5.45	4.24	8.45	8.18
b21	4.27	5.00	4.46	4.46	5.71	5.71	4.63	3.59	11.11	10.81
b22	4.49	4.17	5.05	5.05	4.72	4.72	4.79	3.47	7.34	6.97

Table A.38: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T6 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
B01	7.87	7.72	7.87	7.72	7.72	7.72	9.93	7.87	15.64	13.05
B02	9.26	9.26	12.31	12.31	9.26	9.26	13.17	9.26	15.36	15.36
B03	4.92	4.82	8.15	8.15	8.15	6.53	8.10	6.44	11.29	11.29
B04	4.23	3.63	4.23	3.89	3.35	3.35	3.58	2.72	8.98	8.84
B05	1.81	1.59	1.80	1.80	1.83	1.83	1.82	1.22	5.77	5.49
B06	12.31	9.26	9.26	9.26	9.26	9.26	13.76	9.26	12.13	12.13
B07	4.19	3.64	4.89	4.89	4.06	4.03	4.06	3.29	7.65	6.56
B08	6.25	6.25	8.26	8.26	6.13	6.13	8.12	6.13	10.39	8.08
B09	10.78	10.78	9.00	9.00	10.78	10.78	10.78	7.21	14.35	12.57
B10	6.86	6.86	6.78	6.78	6.86	6.86	8.44	6.86	9.51	9.51
B11	1.93	1.93	2.34	2.34	2.34	2.07	3.01	2.43	4.47	4.47
B12	4.99	4.84	4.11	4.11	4.94	4.94	4.25	3.39	7.42	6.59
B13	1.27	1.27	1.70	1.70	2.05	1.95	2.93	1.95	7.35	5.29
B14	2.00	1.99	2.37	2.37	3.14	3.13	2.54	1.84	6.54	6.30
B17	0.86	0.84	1.17	1.12	1.45	1.45	0.98	0.86	5.02	4.91
B18	0.20	0.20	0.21	0.21	0.33	0.33	0.26	0.21	4.16	4.16
B19	0.28	0.28	0.28	0.28	0.21	0.21	0.29	0.26	4.86	4.86
B20	2.02	1.94	2.48	2.48	2.48	2.48	1.90	1.54	6.79	6.38
B21	2.08	2.08	1.70	1.70	2.23	2.23	1.87	1.40	5.91	5.71
B22	2.08	2.05	1.83	1.83	1.45	1.45	1.59	1.31	6.23	6.23

Table A.39: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T1 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	0.62	0.61	1.66	1.62	3.91	3.91	0.49	0.03	13.06	13.06
mnist	9.02	5.97	5.97	4.54	10.83	9.77	4.97	2.45	17.86	17.86
mobilnet1	1.82	1.82	1.72	1.72	6.78	6.78	1.83	1.10	10.08	10.08
OneCore	1.25	1.25	0.11	0.11	2.40	2.20	0.47	0.01	10.37	7.64
PuLSAR	0.49	0.49	0.09	0.09	3.82	3.82	0.39	0.08	11.30	11.30
WasgaServer	0.03	0.03	0.67	0.67	4.92	4.92	0.38	0.01	13.19	13.19
bitonic_mesh	0.71	0.71	1.42	1.42	12.70	12.70	0.71	0.71	12.07	12.07
cholesky_bdti	1.90	1.13	1.90	1.90	4.83	4.83	2.10	0.97	10.96	10.96
dart	1.97	1.97	1.69	1.69	2.79	2.79	2.75	1.41	6.71	6.71
denoise	0.00	0.00	0.00	0.00	0.07	0.07	0.00	0.00	6.85	6.85
des90	2.94	2.15	1.42	1.42	11.27	11.27	0.95	0.58	12.21	12.21
xge_mac	2.50	2.50	3.98	3.98	6.77	6.77	4.28	3.98	8.67	8.59
cholesky_mc	1.08	1.08	2.00	1.40	4.88	4.88	1.67	0.97	12.87	12.65

Table A.40: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T2 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	0.84	0.73	2.08	2.08	3.92	3.92	0.80	0.36	20.92	20.92
mnist	9.02	5.97	5.97	4.54	17.49	16.24	5.16	2.73	30.71	25.95
mobilnet1	1.82	1.82	2.82	2.82	7.99	7.99	1.83	1.10	10.01	10.01
OneCore	1.25	1.25	1.72	1.70	4.25	4.09	0.74	0.01	17.67	14.69
PuLSAR	0.49	0.49	1.12	1.12	7.13	7.13	0.77	0.13	19.04	19.04
WasgaServer	0.44	0.44	2.26	2.26	5.74	5.74	0.56	0.01	23.91	23.91
bitonic_mesh	1.34	0.71	4.46	4.46	18.66	18.66	1.43	1.34	18.83	18.83
cholesky_bdti	4.67	2.82	2.00	2.00	6.68	6.68	2.49	1.90	19.89	19.89
dart	1.97	1.97	2.25	2.25	2.80	2.80	3.29	1.41	8.68	8.68
denoise	0.00	0.00	0.00	0.00	0.07	0.07	0.00	0.00	7.13	7.13
des90	2.94	2.15	1.47	1.47	13.20	13.20	1.62	1.34	18.97	18.97
xge_mac	2.50	2.50	5.12	5.12	9.90	9.81	4.28	3.98	20.59	14.59
cholesky_mc	2.11	2.11	2.00	2.00	5.65	4.88	2.87	1.90	21.73	21.73

Table A.41: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T3 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	0.15	0.08	0.46	0.46	1.61	1.61	0.08	0.01	8.68	8.68
mnist	4.17	2.92	2.92	2.92	6.53	6.44	2.78	1.30	11.11	11.11
mobilnet1	1.82	1.82	1.72	1.72	3.11	3.11	1.83	1.10	6.34	6.34
OneCore	0.32	0.32	0.11	0.11	1.96	1.95	0.17	0.01	6.60	5.26
PuLSAR	0.25	0.25	0.08	0.08	2.61	2.61	0.10	0.01	7.78	7.78
WasgaServer	0.01	0.01	0.47	0.47	2.56	2.56	0.08	0.01	7.82	7.82
bitonic_mesh	0.03	0.03	1.42	1.42	8.14	8.14	0.03	0.03	9.03	9.03
cholesky_bdti	0.97	0.97	0.97	0.97	3.25	3.25	0.90	0.21	6.18	6.18
dart	1.41	1.41	1.69	1.69	1.40	1.40	2.14	1.13	6.72	6.72
denoise	0.00	0.00	0.00	0.00	0.07	0.07	0.00	0.00	1.36	1.36
des90	1.42	1.42	1.42	1.42	7.20	7.20	0.18	0.03	8.22	8.22
xge_mac	2.50	2.50	3.98	3.98	5.72	5.72	4.28	3.98	7.02	6.94
cholesky_mc	1.08	1.08	1.13	1.13	3.09	3.09	0.92	0.26	7.98	7.98

Table A.42: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T4 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	0.62	0.62	3.21	3.21	8.36	8.36	2.02	0.85	16.55	16.55
mnist	11.01	11.01	9.39	9.39	19.29	19.20	9.81	5.88	22.06	22.06
mobilnet1	5.35	5.35	4.95	4.95	12.57	12.57	4.19	1.83	13.05	13.05
OneCore	1.25	1.16	1.88	1.88	6.34	5.31	2.12	1.56	14.00	10.29
PuLSAR	2.92	2.92	2.20	2.20	8.68	8.68	2.23	1.07	28.98	28.98
WasgaServer	0.93	0.93	0.77	0.77	9.06	9.06	0.98	0.02	20.40	20.40
bitonic_mesh	2.10	2.10	2.94	2.94	24.21	24.21	2.21	2.10	16.60	16.60
cholesky_bdti	2.98	2.98	4.67	3.85	20.76	20.76	4.03	2.82	18.25	18.25
dart	3.93	3.93	4.21	4.21	5.89	5.89	4.71	3.37	11.50	11.50
denoise	0.00	0.00	0.02	0.02	1.80	1.80	0.00	0.00	5.55	5.55
des90	2.94	2.23	2.94	2.94	18.88	18.88	2.47	2.23	19.68	19.68
xge_mac	9.55	9.55	9.55	9.55	18.86	18.77	10.49	5.46	20.42	17.81
cholesky_mc	2.82	2.82	6.62	6.57	15.32	15.32	3.98	2.11	18.30	18.30

Table A.43: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T5 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	0.99	0.99	3.83	3.83	8.57	8.57	2.33	0.85	29.35	29.35
mnist	12.63	12.63	12.54	12.54	28.81	25.48	14.24	9.30	33.76	28.91
mobilnet1	7.22	7.22	6.75	6.75	21.93	21.93	4.88	3.08	18.74	18.74
OneCore	2.18	2.06	2.24	2.24	5.39	5.13	2.76	1.54	10.37	9.77
PuLSAR	2.89	2.89	1.40	1.40	7.91	7.91	2.71	1.90	18.27	18.27
WasgaServer	0.84	0.84	1.06	1.06	14.21	14.21	1.18	0.43	24.87	24.87
bitonic_mesh	2.94	2.94	4.46	4.46	39.47	39.47	2.35	2.10	31.01	31.01
cholesky_bdti	4.67	2.98	4.67	3.74	19.78	19.78	4.22	2.82	16.46	16.46
dart	5.60	5.60	5.04	5.04	6.29	6.29	5.80	4.21	12.31	12.31
denoise	0.00	0.00	0.03	0.03	3.21	3.21	0.00	0.00	12.17	12.17
des90	3.67	3.67	4.46	4.46	19.24	18.53	2.34	2.23	18.92	18.92
xge_mac	13.28	13.20	13.20	12.86	21.64	20.25	14.45	8.42	29.64	20.17
cholesky_mc	4.78	4.78	6.62	5.70	17.17	17.17	4.52	2.82	16.46	16.46

Table A.44: Results for critical path: $d^{\Pi}_{\max}(H)$, on target T6 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	0.01	0.01	1.12	1.12	2.67	2.67	0.20	0.02	9.74	9.74
mnist	4.54	4.54	4.54	4.45	9.39	9.30	5.35	4.54	11.01	9.77
mobilnet1	2.23	2.23	1.83	1.83	6.34	6.34	1.92	0.58	5.72	5.72
OneCore	0.41	0.32	0.29	0.29	2.39	2.39	0.70	0.23	11.04	8.55
PuLSAR	0.44	0.44	0.21	0.21	2.58	2.58	0.56	0.48	9.63	9.63
WasgaServer	0.01	0.01	0.01	0.01	4.32	4.32	0.08	0.01	7.24	7.24
bitonic_mesh	1.42	0.66	1.42	1.42	12.83	12.83	0.09	0.03	9.79	9.79
cholesky_bdti	0.97	0.97	2.00	2.00	7.82	7.82	0.92	0.21	8.96	8.96
dart	2.53	2.53	2.25	2.25	2.28	2.28	2.78	2.25	7.28	7.28
denoise	0.00	0.00	0.01	0.01	1.65	1.65	0.00	0.00	1.53	1.53
des90	1.42	1.42	1.42	1.42	8.18	8.18	0.24	0.03	9.61	9.61
xge_mac	4.41	4.32	4.41	4.32	6.85	6.85	5.18	3.98	8.59	7.11
cholesky_mc	1.08	1.08	1.13	1.08	6.08	6.08	1.02	0.97	7.11	7.11

Connectivity cost

Results on connectivity cost presented in Chapter 6 are based on tables introduced in this subsection. Each table shows us effect of DKFM on connectivity cost of partition produced by min-cut algorithms. These results have been used to make the figures presenting the relative connectivity cost of each partition. The following figures corresponds to the complementary results of the one presented in Chapter 6.

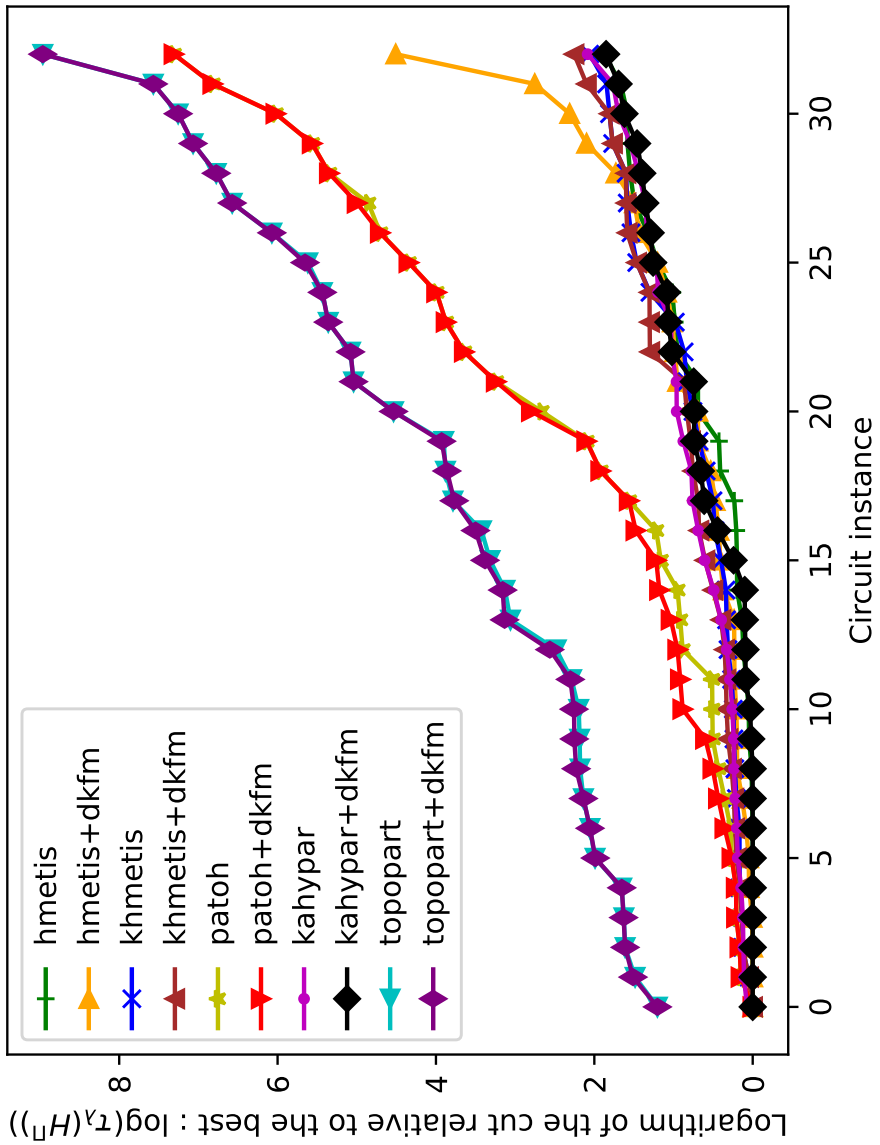


Figure A.1: Connectivity cost relative to the best in a logarithmic scale on target topology T1 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFM. Similar to results on target topology T3, DKFM improves faintly some partitions produced by KaHyPar. However, for hMetis, khMetis, PaToH, and TopoPart, DKFM slightly reduces connectivity costs.

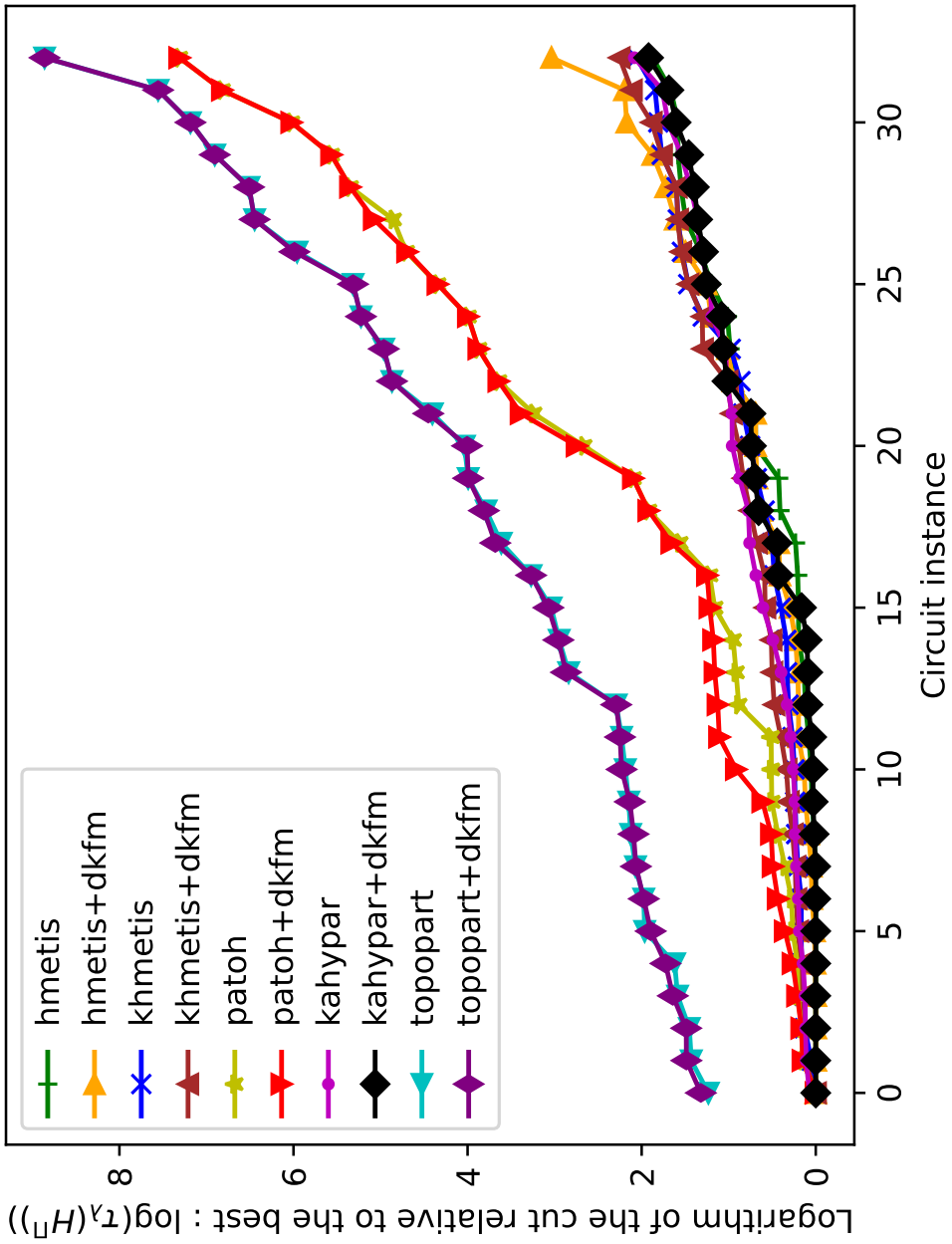


Figure A.2: Connectivity cost relative to the best in a logarithmic scale on target topology T2 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFM.

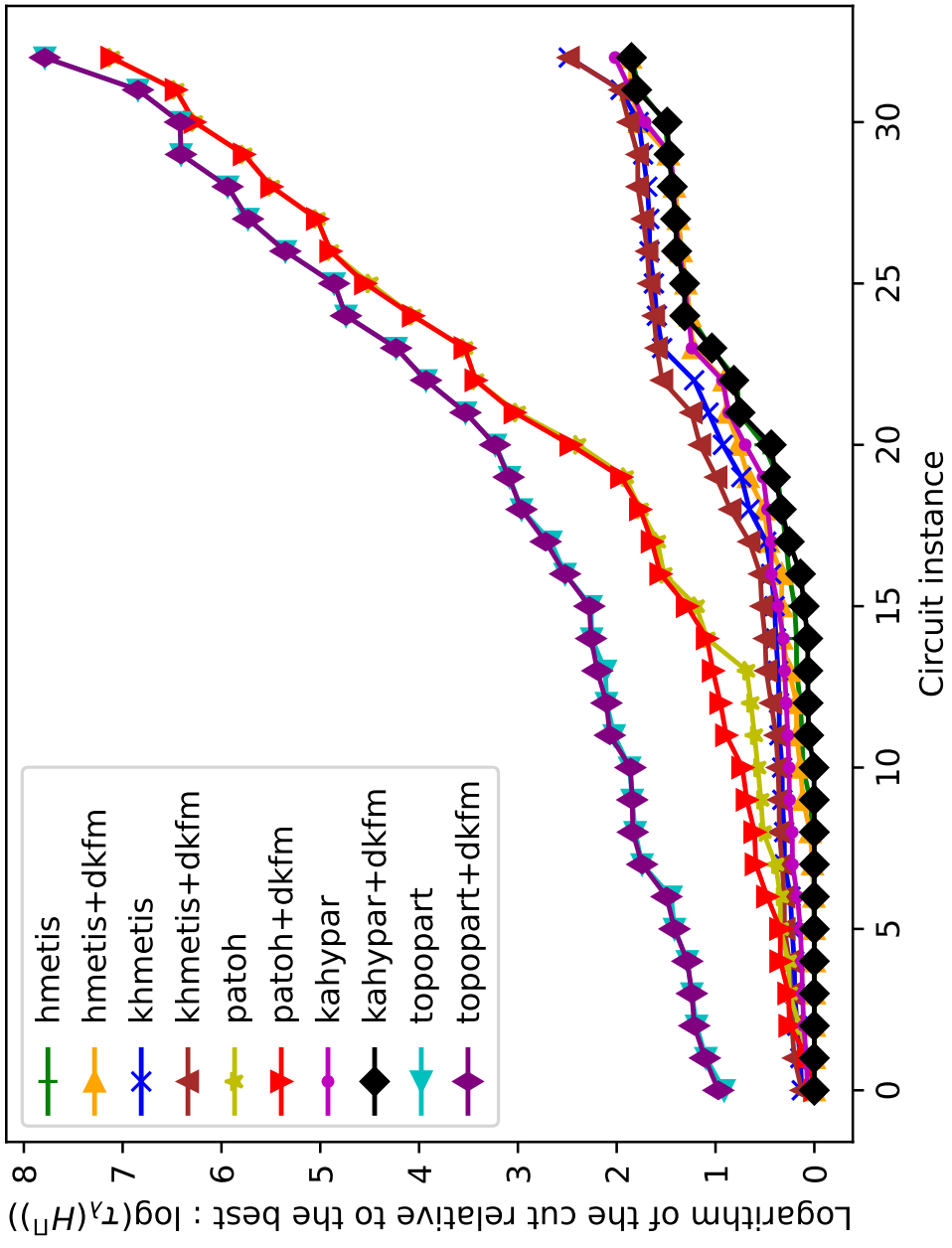


Figure A.3: Connectivity cost relative to the best in a logarithmic scale on target topology T4 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFM.

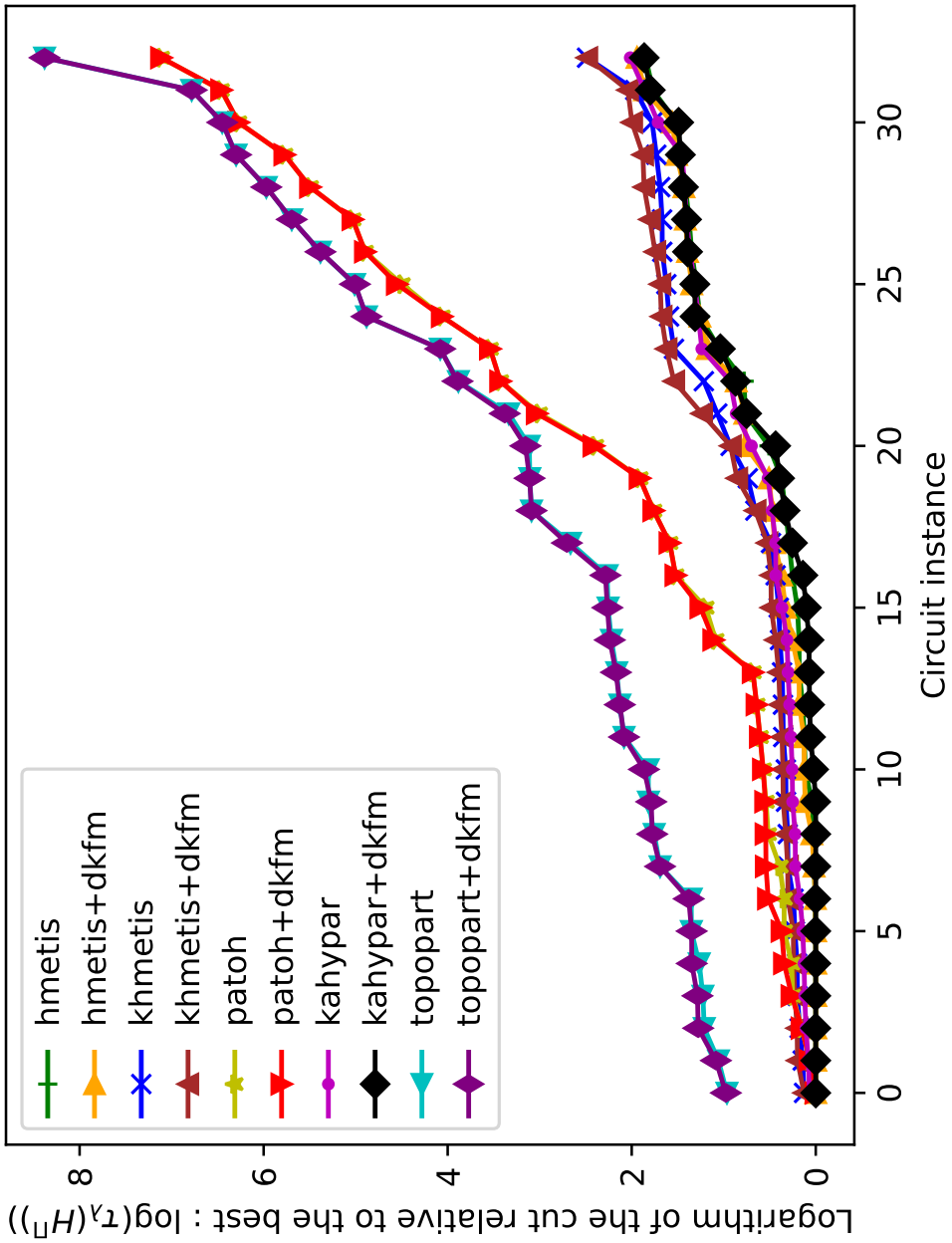


Figure A.4: Connectivity cost relative to the best in a logarithmic scale on target topology T5 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFM.

Table A.45: Results for connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T1 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
B01	15.78	17.56	19.56	20.78	20.04	20.04	14.15	15.26	30.78	32.78
B02	14.31	17.91	17.22	17.91	17.91	17.91	12.82	14.00	19.44	20.44
B03	18.54	18.54	29.96	29.96	22.10	30.34	18.09	19.21	58.73	58.73
B04	43.17	43.17	43.14	43.14	50.32	50.44	44.10	41.02	214.39	214.39
B05	33.43	33.43	37.96	37.96	39.38	39.38	41.72	34.53	329.00	334.41
B06	18.91	21.91	23.75	23.75	26.57	26.57	18.24	19.66	30.40	28.41
B07	40.74	50.76	50.28	50.28	52.29	52.29	45.16	40.14	133.72	133.72
B08	29.97	29.97	35.70	35.70	36.40	36.40	29.89	31.46	66.35	68.10
B09	24.32	24.32	23.43	32.47	29.83	29.83	24.72	26.04	79.37	84.19
B10	35.58	35.58	37.90	37.90	46.03	46.03	32.68	34.31	90.65	90.97
B11	45.56	66.14	58.16	64.93	60.17	82.88	52.72	58.00	228.19	228.19
B12	40.71	61.50	42.73	66.01	44.63	44.63	40.93	39.94	321.71	344.11
B13	2.51	7.44	5.76	5.76	6.26	6.26	7.00	7.26	76.95	83.05
B14	228.53	228.53	309.84	309.84	375.90	375.90	293.80	226.81	2709.14	2939.55
B17	153.01	245.69	187.88	187.88	236.40	236.40	227.89	155.97	7569.73	7775.84
B18	100.63	100.63	141.06	141.06	143.35	143.35	241.69	156.99	16153.86	16153.86
B19	375.41	375.41	286.12	286.12	145.29	145.29	380.08	306.03	33293.39	33293.39
B20	313.76	313.76	300.33	300.33	425.78	716.51	339.82	255.03	5434.42	5851.04
B21	296.62	296.62	297.89	297.89	391.46	764.62	284.08	235.43	5367.37	5530.10
B22	365.41	365.41	354.11	596.50	348.22	348.22	366.18	300.11	8258.14	8805.05

Table A.46: Results for connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T2 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	15.78	18.78	19.56	19.78	20.04	22.04	14.15	15.26	30.00	28.00
b02	14.31	14.31	17.22	18.22	17.91	17.91	12.82	15.00	18.83	17.83
b03	18.54	24.29	29.96	29.96	22.10	22.10	18.09	19.21	64.44	64.44
b04	43.17	43.17	43.14	73.38	50.32	67.18	44.10	41.02	175.29	181.65
b05	33.43	33.43	37.96	37.96	39.38	39.38	41.72	34.53	241.12	241.12
b06	18.91	21.83	23.75	25.66	26.57	26.57	18.24	19.35	32.62	30.71
b07	40.74	40.74	50.28	50.28	52.29	52.29	45.16	40.14	137.84	150.29
b08	29.97	29.97	35.70	41.50	36.40	45.51	29.89	31.46	73.46	84.43
b09	24.32	24.32	23.43	32.47	29.83	29.83	24.72	26.04	78.62	83.51
b10	35.58	35.58	37.90	37.90	46.03	54.94	32.68	34.35	90.11	92.02
b11	45.56	45.56	58.16	75.90	60.17	83.98	52.72	47.74	190.32	201.38
b12	40.71	40.71	42.73	95.08	44.63	44.72	40.93	39.94	286.72	306.18
b13	2.51	4.93	5.76	5.76	6.26	6.26	7.00	7.26	65.96	65.96
b14	228.53	228.53	309.84	309.84	375.90	378.55	293.80	233.67	2243.78	2243.78
b17	153.01	205.77	187.88	187.88	236.40	236.40	227.89	155.97	5671.79	6078.86
b18	100.63	100.63	141.06	141.06	143.35	143.35	241.69	156.99	8231.00	8627.89
b19	375.41	375.41	286.12	286.12	145.29	145.29	380.08	306.03	20779.54	20779.54
b20	313.76	313.76	300.33	300.33	425.78	887.80	339.82	255.03	4861.80	4905.74
b21	296.62	296.62	297.89	297.89	391.46	762.76	284.08	235.43	4021.67	4145.30
b22	365.41	365.41	354.11	354.11	348.22	348.22	366.18	299.81	6264.11	6470.19

Table A.47: Results for connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T3 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
B01	15.78	17.56	19.56	19.56	20.04	20.04	14.15	17.26	24.04	24.04
B02	14.31	14.31	17.22	17.22	17.91	17.91	12.82	14.00	18.75	20.75
B03	18.54	18.54	29.96	29.96	22.10	22.10	18.09	19.21	40.82	46.44
B04	43.17	43.17	43.14	43.14	50.32	54.90	44.10	41.02	120.42	120.42
B05	33.43	33.43	37.96	37.96	39.38	39.38	41.72	34.53	206.37	206.37
B06	18.91	21.91	23.75	23.75	26.57	26.57	18.24	19.35	25.96	27.96
B07	40.74	40.74	50.28	50.28	52.29	52.29	45.16	40.14	121.84	133.86
B08	29.97	29.97	35.70	35.70	36.40	42.51	29.89	31.46	55.51	55.51
B09	24.32	24.32	23.43	26.47	29.83	29.83	24.72	26.04	53.26	53.26
B10	35.58	35.58	37.90	37.90	46.03	46.03	32.68	34.31	76.66	76.66
B11	45.56	76.77	58.16	73.31	60.17	83.98	52.72	51.08	132.00	132.00
B12	40.71	63.25	42.73	65.34	44.63	44.63	40.93	39.94	194.36	212.00
B13	2.51	2.51	5.76	5.76	6.26	9.35	7.00	7.26	55.84	64.19
B14	228.53	228.53	309.84	309.84	375.90	375.90	293.80	226.81	2303.83	2334.47
B17	153.01	245.69	187.88	187.88	236.40	236.40	227.89	155.97	4440.85	4681.71
B18	100.63	100.63	141.06	141.06	143.35	143.35	241.69	156.99	6535.17	6535.17
B19	375.41	375.41	286.12	286.12	145.29	145.29	380.08	306.03	14537.49	14537.49
B20	313.76	313.76	300.33	300.33	425.78	822.86	339.82	255.03	3565.17	3854.48
B21	296.62	296.62	297.89	297.89	391.46	764.62	284.08	235.19	3598.38	3644.19
B22	365.41	365.41	354.11	354.11	348.22	348.22	366.18	301.02	5974.03	6141.00

Table A.48: Results for connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T4 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	25.52	25.52	34.26	34.26	34.04	36.04	25.93	28.04	39.78	39.78
b02	24.22	24.53	27.13	27.13	25.83	26.91	21.18	23.05	31.27	31.27
b03	34.58	34.58	52.04	58.61	45.01	46.60	33.86	36.52	94.02	94.02
b04	74.75	74.75	91.11	91.11	104.95	104.95	79.66	74.38	245.00	250.68
b05	63.77	63.77	73.49	108.50	74.20	102.72	75.66	63.82	230.59	230.59
b06	30.88	30.88	39.88	39.88	41.10	41.10	29.31	30.71	44.54	44.54
b07	65.31	65.31	89.41	89.41	92.45	113.09	67.89	62.58	155.88	165.26
b08	51.39	51.39	64.64	72.20	61.17	61.17	47.78	47.78	106.72	106.72
b09	44.57	53.44	52.86	54.30	51.76	56.90	39.74	44.18	101.51	100.76
b10	62.66	71.75	72.61	77.61	85.34	85.34	57.91	61.03	118.47	118.51
b11	73.52	100.48	99.79	124.80	92.82	92.82	81.99	77.15	216.28	218.62
b12	81.81	100.49	86.01	93.03	93.82	93.82	81.46	80.06	307.94	322.32
b13	9.19	9.19	14.95	14.95	17.45	25.63	15.43	13.61	69.56	72.73
b14	437.98	437.98	463.58	625.29	637.36	982.96	513.79	380.15	2330.99	2387.15
b17	457.66	457.66	563.98	563.98	538.76	538.76	522.56	382.38	5497.33	5800.49
b18	273.98	273.98	390.12	390.12	292.30	292.30	395.37	290.07	5305.66	5305.66
b19	523.60	523.60	479.48	479.48	376.70	376.70	605.31	485.28	12863.15	12863.15
b20	630.08	920.84	653.96	653.96	811.17	1151.87	607.30	503.36	3908.89	4245.80
b21	643.99	643.99	626.36	626.36	884.25	1034.04	624.57	501.28	4752.89	4861.88
b22	779.01	1299.59	803.10	1706.53	983.40	983.40	693.72	537.02	6691.90	6691.90

Table A.49: Results for connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T5 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
B01	25.52	28.04	34.26	37.26	34.04	34.04	25.93	28.04	37.78	37.78
B02	24.22	24.53	27.13	28.13	25.83	25.83	21.18	23.05	30.66	30.66
B03	34.58	34.58	52.04	56.28	45.01	45.01	33.86	36.52	85.37	86.07
B04	74.75	74.75	91.11	91.11	104.95	128.09	79.66	74.38	263.78	268.95
B05	63.77	63.77	73.49	108.04	74.20	74.20	75.66	63.82	214.16	245.65
B06	30.88	33.57	39.88	42.80	41.10	41.10	29.31	30.71	43.54	42.15
B07	65.31	65.31	89.41	89.41	92.45	108.17	67.89	62.58	164.29	166.24
B08	51.39	51.39	64.64	72.72	61.17	61.17	47.78	47.78	105.33	110.85
B09	44.57	50.90	52.86	54.04	51.76	54.76	39.74	44.18	96.54	101.37
B10	62.66	68.30	72.61	72.61	85.34	87.43	57.91	61.03	111.19	110.74
B11	73.52	82.04	99.79	117.18	92.82	104.53	81.99	77.55	207.91	213.74
B12	81.81	81.81	86.01	92.15	93.82	93.82	81.46	80.00	279.43	287.28
B13	9.19	9.19	14.95	14.95	17.45	17.45	15.43	13.61	77.21	77.21
B14	437.98	609.18	463.58	463.58	637.36	637.36	513.79	409.04	2198.62	2247.59
B17	457.66	457.66	563.98	563.98	538.76	538.76	522.56	391.49	5495.99	5715.91
B18	273.98	273.98	390.12	390.12	292.30	292.30	395.37	290.07	6015.34	6015.34
B19	523.60	523.60	479.48	479.48	376.70	376.70	605.31	485.28	8424.31	8836.47
B20	630.08	1021.52	653.96	653.96	811.17	811.17	607.30	472.31	4512.30	4545.84
B21	643.99	643.99	626.36	626.36	884.25	884.25	624.57	499.28	4597.18	4681.47
B22	779.01	779.01	803.10	803.10	983.40	983.40	693.72	537.02	5181.18	5279.24

Table A.50: Results for connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T6 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	25.52	28.78	34.26	35.26	34.04	34.04	25.93	28.04	45.78	43.78
b02	24.22	24.22	27.13	27.13	25.83	25.83	21.18	23.05	33.57	33.57
b03	34.58	34.58	52.04	54.98	45.01	48.57	33.86	36.52	73.38	73.38
b04	74.75	74.75	91.11	91.11	104.95	104.95	79.66	74.38	245.08	262.24
b05	63.77	63.77	73.49	108.04	74.20	74.20	75.66	63.82	250.08	273.08
b06	30.88	30.88	39.88	40.88	41.10	41.10	29.31	30.71	41.06	41.06
b07	65.31	65.31	89.41	89.41	92.45	115.07	67.89	62.58	142.33	158.63
b08	51.39	51.39	64.64	72.41	61.17	61.17	47.78	47.78	83.96	89.96
b09	44.57	44.57	52.86	52.86	51.76	51.76	39.74	43.29	86.54	96.54
b10	62.66	64.66	72.61	72.61	85.34	85.34	57.91	61.03	99.57	99.57
b11	73.52	88.87	99.79	122.16	92.82	100.94	81.99	72.22	225.17	225.17
b12	81.81	81.81	86.01	92.15	93.82	93.82	81.46	80.06	290.53	299.27
b13	9.19	9.19	14.95	14.95	17.45	26.72	15.43	13.61	87.31	92.91
b14	437.98	437.98	463.58	581.28	637.36	1036.61	513.79	380.15	2363.41	2407.88
b17	457.66	457.66	563.98	563.98	538.76	538.76	522.56	391.49	5729.51	5765.87
b18	273.98	273.98	390.12	390.12	292.30	292.30	395.37	290.07	8302.64	8302.64
b19	523.60	523.60	479.48	479.48	376.70	376.70	605.31	485.28	16437.47	16437.47
b20	630.08	691.91	653.96	653.96	811.17	811.17	607.30	472.31	4740.33	4999.19
b21	643.99	643.99	626.36	626.36	884.25	884.25	624.57	499.28	4347.59	4385.01
b22	779.01	1106.05	803.10	803.10	983.40	983.40	693.72	537.02	6241.21	6241.21

Table A.51: Results for connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T1 for circuits in Chipyard and Titan sets.

Instance	nMetis	+DKFM	kHMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	339.37	427.52	546.69	1246.74	16334.62	16334.62	471.18	375.28	72204.32	72204.32
mnist	59.13	190.47	76.93	115.76	1558.20	1798.81	47.08	49.01	5266.51	5266.51
mobilnet1	124.96	124.96	134.81	134.81	21324.92	21324.92	174.56	171.18	71450.56	71450.56
OneCore	291.40	291.40	354.53	354.53	3662.96	4178.36	284.47	254.86	11224.67	11040.37
PuLSAR	550.20	550.20	673.69	673.69	24496.98	24496.98	738.65	451.50	69758.00	69758.00
WasgaServer	433.53	433.53	565.71	565.71	91152.84	91152.84	551.00	475.16	310204.31	310204.31
bitonic_mesh	211.71	211.71	231.62	231.62	71051.11	71051.11	180.63	184.54	92041.82	92041.82
cholesky_bdti	309.30	5311.50	289.50	289.50	24165.87	24165.87	318.99	318.45	68608.60	68608.60
dart	231.19	231.19	220.92	220.92	8449.84	8449.84	279.80	241.65	20604.75	20604.75
denoise	8.62	8.62	9.08	9.08	611.92	611.92	63.91	7.95	62171.28	62171.28
des90	172.75	370.85	215.51	215.51	33674.90	33674.90	155.23	158.30	52052.23	52052.23
xge_mac	84.21	84.21	92.80	92.80	647.48	647.48	83.70	89.10	1226.54	1204.79
cholesky_mc	170.30	170.30	203.21	247.03	12474.93	12474.93	244.88	208.44	31098.36	32204.75

Table A.52: Results for connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T2 for circuits in Chipyard and Titan sets.

Instance	nMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	339.37	630.49	546.69	546.69	16334.62	16334.62	471.18	399.64	69043.37	69043.37
mnist	59.13	61.07	76.93	115.76	1558.20	1943.67	47.08	49.01	4701.40	4848.31
mobilnet1	124.96	124.96	134.81	134.81	21324.92	21324.92	174.56	125.39	54896.87	54896.87
OneCore	291.40	291.40	354.53	616.48	3662.96	3966.80	284.47	254.86	11244.26	11539.07
PuLSAR	550.20	550.20	673.69	673.69	24496.98	24496.98	738.65	451.50	58753.23	58753.23
WasgaServer	433.53	433.53	565.71	565.71	91152.84	91152.84	551.00	475.16	273364.43	273364.43
bitonic_mesh	211.71	416.72	231.62	231.62	71051.11	71051.11	180.63	184.54	90752.78	90752.78
cholesky_bdti	309.30	1218.92	289.50	289.50	24165.87	24165.87	318.99	316.22	58323.14	58323.14
dart	231.19	231.19	220.92	220.92	8449.84	8449.84	279.80	241.65	11980.28	11980.28
denoise	8.62	8.62	9.08	9.08	611.92	611.92	63.91	7.95	55994.82	55994.82
des90	172.75	331.08	215.51	215.51	33674.90	33674.90	155.23	158.30	48460.31	48460.31
xge_mac	84.21	84.21	92.80	92.80	647.48	750.48	83.70	89.10	1433.64	1391.04
cholesky_mc	170.30	170.30	203.21	203.21	12474.93	12318.01	244.88	227.31	20981.25	20981.25

Table A.53: Results for connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T3 for circuits in Chipyard and Titan sets.

Instance	nMetis	+DKFM	kahMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	339.37	417.92	546.69	546.69	16334.62	16334.62	471.18	375.28	40149.34	40149.34
mnist	59.13	190.47	76.93	76.93	1558.20	1777.23	47.08	49.01	3929.59	3929.59
mobilnet1	124.96	124.96	134.81	134.81	21324.92	21324.92	174.56	171.18	57561.03	57561.03
OneCore	291.40	291.40	354.53	354.53	3662.96	3981.09	284.47	254.86	8019.80	8074.33
PuLSAR	550.20	550.20	673.69	673.69	24496.98	24496.98	738.65	451.50	50737.17	50737.17
WasgaServer	433.53	433.53	565.71	565.71	91152.84	91152.84	551.00	475.16	121510.73	121510.73
bitonic_mesh	211.71	211.71	231.62	231.62	71051.11	71051.11	180.63	184.54	43656.45	43656.45
cholesky_bdti	309.30	309.30	289.50	289.50	24165.87	24165.87	318.99	307.08	34461.86	34461.86
dart	231.19	231.19	220.92	220.92	8449.84	8449.84	279.80	241.65	10362.24	10589.51
denoise	8.62	8.62	9.08	9.08	611.92	611.92	63.91	7.95	2576.59	2576.59
des90	172.75	172.75	215.51	215.51	33674.90	33674.90	155.23	158.30	23958.57	23958.57
xge_mac	84.21	84.21	92.80	92.80	647.48	647.48	83.70	89.10	733.54	871.67
cholesky_mc	170.30	170.30	203.21	203.21	12474.93	12474.93	244.88	227.31	16160.26	16160.26

Table A.54: Results for connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T4 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	553.49	553.49	1066.99	1066.99	32142.63	32142.63	724.25	636.81	63339.58	63339.58
mnist	144.72	144.72	203.84	203.84	3370.12	3540.75	153.08	158.31	4850.16	4850.16
mobilnet1	261.15	261.15	353.53	353.53	34894.44	34894.44	402.84	364.09	80351.36	80351.36
OneCore	697.72	908.33	884.14	884.14	6266.75	6735.79	712.94	571.48	12495.50	12598.32
PuLSAR	1506.55	1506.55	1813.54	1813.54	38721.46	38721.46	1670.17	1257.11	86575.31	86575.31
WasgaServer	752.61	752.61	907.82	907.82	163202.56	163202.56	773.57	671.19	254302.30	254302.30
bitonic_mesh	441.10	441.10	499.45	499.45	131346.60	131346.60	383.18	397.26	100498.55	100498.55
cholesky_bdti	667.28	667.28	672.46	678.65	60250.82	60250.82	731.40	723.30	69748.81	69748.81
dart	397.82	397.82	443.32	443.32	13440.29	13440.29	434.29	390.46	19873.76	19873.76
denoise	14.11	14.11	169.44	169.44	4536.17	4536.17	105.93	15.09	34050.75	34050.75
des90	379.46	536.14	423.98	423.98	53545.13	53545.13	357.38	370.87	50801.03	50801.03
xge_mac	222.31	222.31	297.86	297.86	1133.25	1164.18	220.02	228.51	1414.97	1420.26
cholesky_mc	291.53	291.53	394.86	885.98	20750.47	20750.47	321.48	305.78	28614.20	28614.20

Table A.55: Results for connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T5 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	553.49	553.49	1066.99	1066.99	32142.63	32142.63	724.25	636.81	72999.51	72999.51
mnist	144.72	144.72	203.84	203.84	3370.12	3557.89	153.08	158.31	5642.35	5621.69
mobilnet1	261.15	261.15	353.53	353.53	34894.44	34894.44	402.84	364.09	77503.64	77503.64
OneCore	697.72	893.57	884.14	884.14	6266.75	6386.59	712.94	571.48	12854.15	12812.20
PuLSAR	1506.55	1506.55	1813.54	1813.54	38721.46	38721.46	1670.17	1257.11	74412.45	74412.45
WasgaServer	752.61	752.61	907.82	907.82	163202.56	163202.56	773.57	671.19	26309[illegible]3.96	26309[illegible]3.96
bitonic_mesh	441.10	441.10	499.45	499.45	131346.60	131346.60	383.18	397.26	94376.97	94376.97
cholesky_bdti	667.28	792.57	672.46	734.12	60250.82	60250.82	731.40	733.16	71894.06	71894.06
dart	397.82	397.82	443.32	443.32	13440.29	13440.29	434.29	390.46	18974.90	19[illegible]8.62
denoise	14.11	14.11	169.44	169.44	4536.17	4536.17	105.93	15.09	61577.92	61577.92
des90	379.46	379.46	423.98	423.98	53545.13	53383.46	357.38	370.87	45499.73	45499.73
xge_mac	222.31	235.04	297.86	432.27	1133.25	1161.22	220.02	228.51	1577.67	16[illegible]7.32
cholesky_mc	291.53	291.53	394.86	885.98	20750.47	20750.47	321.48	323.35	29467.61	29467.61

Table A.56: Results for connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T6 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	knMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	553.49	553.49	1066.99	1066.99	32142.63	32142.63	724.25	636.81	48723.13	48723.13
mnist	144.72	144.72	203.84	255.11	3370.12	3446.42	153.08	158.31	4748.70	4974.48
mobilnet1	261.15	261.15	353.53	353.53	34894.44	34894.44	402.84	364.09	66785.10	66785.10
OneCore	697.72	822.05	884.14	884.14	6266.75	6375.35	712.94	571.48	13627.76	14078.18
PuLSAR	1506.55	1506.55	1813.54	1813.54	38721.46	38721.46	1670.17	1257.11	72032.31	72032.31
WasgaServer	752.61	752.61	907.82	907.82	163202.56	163202.56	773.57	671.19	147895.88	147895.88
bitonic_mesh	441.10	873.80	499.45	499.45	131346.60	131346.60	383.18	397.26	46519.73	46519.73
cholesky_bdti	667.28	667.28	672.46	672.46	60250.82	60250.82	731.40	734.65	47576.24	47576.24
dart	397.82	397.82	443.32	443.32	13440.29	13440.29	434.29	390.46	11958.43	11958.43
denoise	14.11	14.11	169.44	169.44	4536.17	4536.17	105.93	15.09	3186.84	3186.84
des90	379.46	379.46	423.98	423.98	53545.13	53545.13	357.38	370.87	40545.63	40545.63
xge_mac	222.31	235.04	297.86	302.12	1133.25	1133.25	220.02	228.51	1125.74	1206.15
cholesky_mc	291.53	291.53	394.86	622.11	20750.47	20750.47	321.48	317.77	19019.96	19019.96

Balance cost

In Chapter 6, we presented a comparison on vertex weight balance of partition. For this purpose, we use the results presented in the following tables. Each table shows us effect of DKFM on balance cost of partition produced by min-cut algorithms. The following figures corresponds to the complementary results of the one presented in Chapter 6.

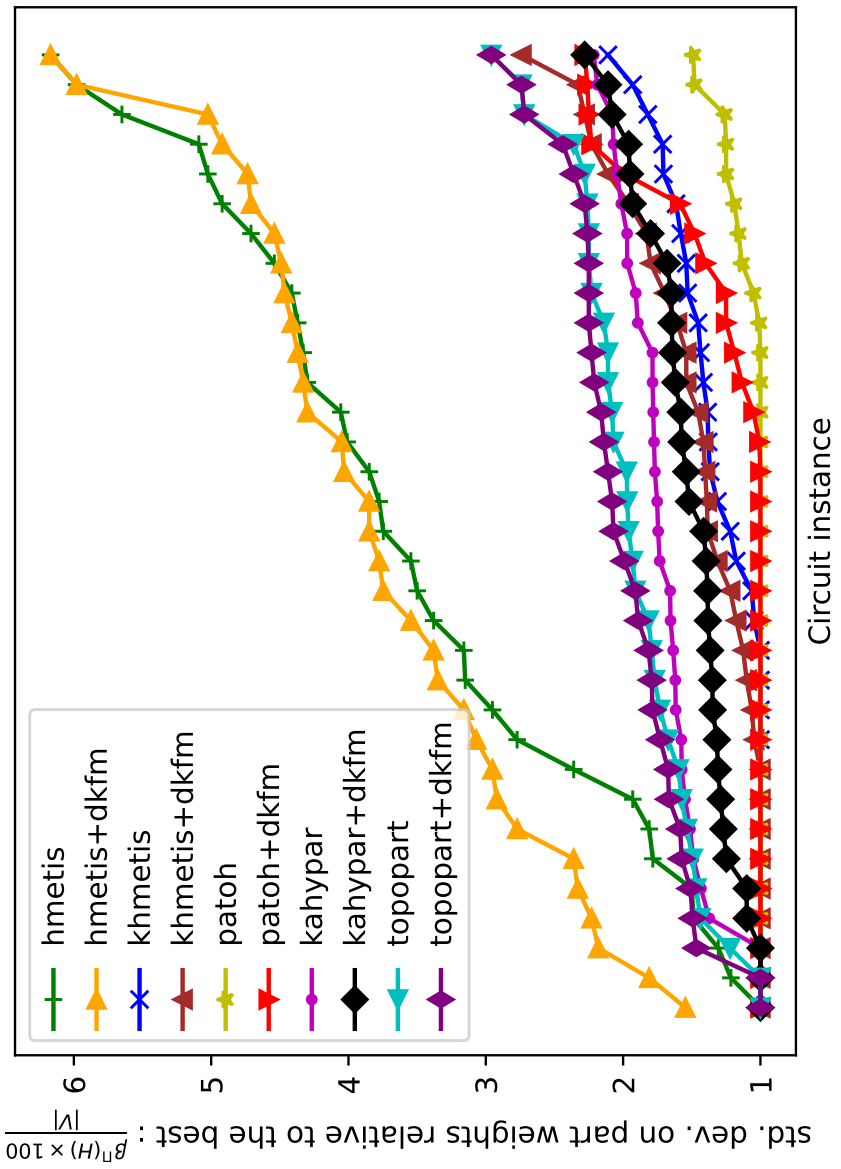


Figure A.5: Balance cost relative to the best on target topology T1 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFM.

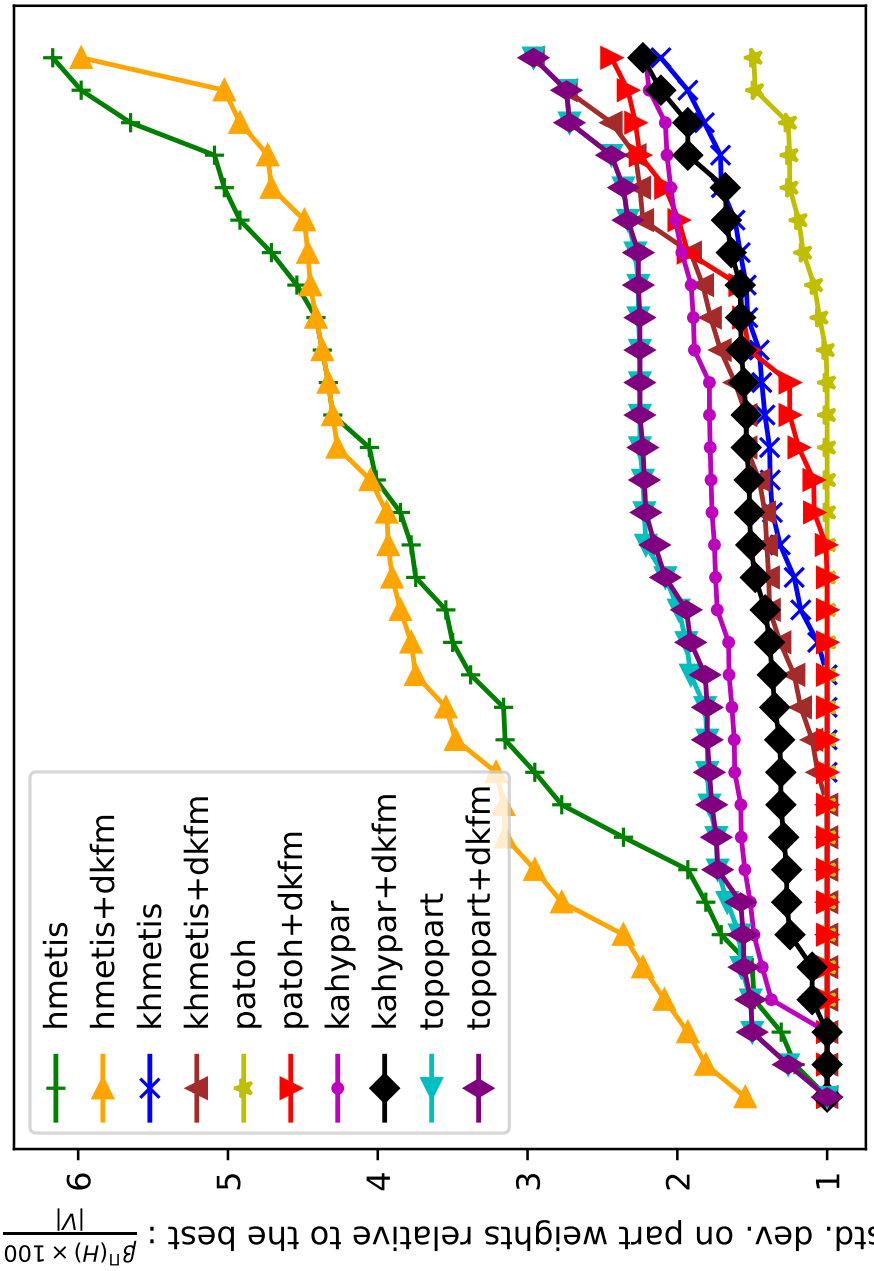


Figure A.6: Balance cost relative to the best on target topology T2 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFM.

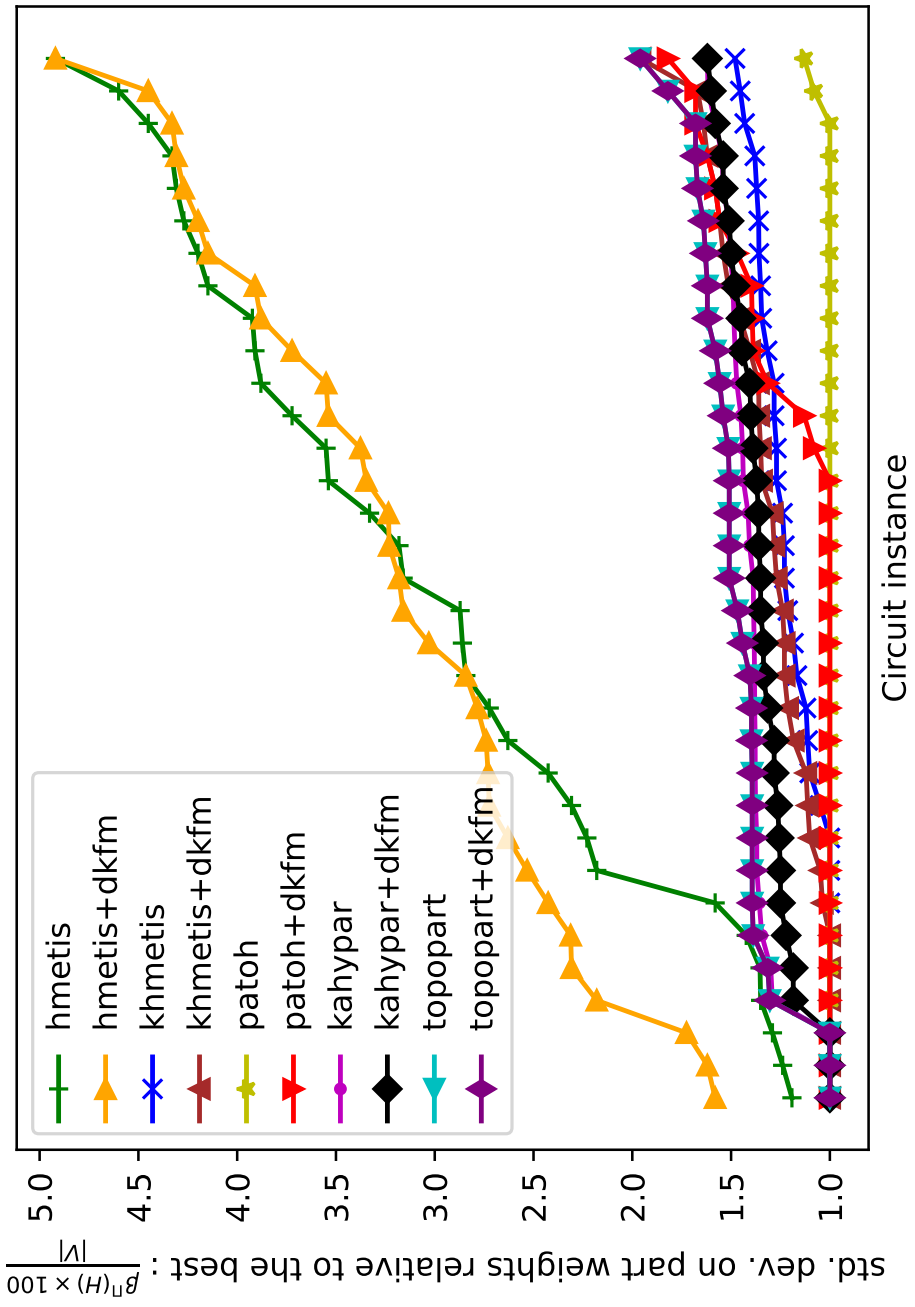


Figure A.7: Balance cost relative to the best on target topology T4 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFM.

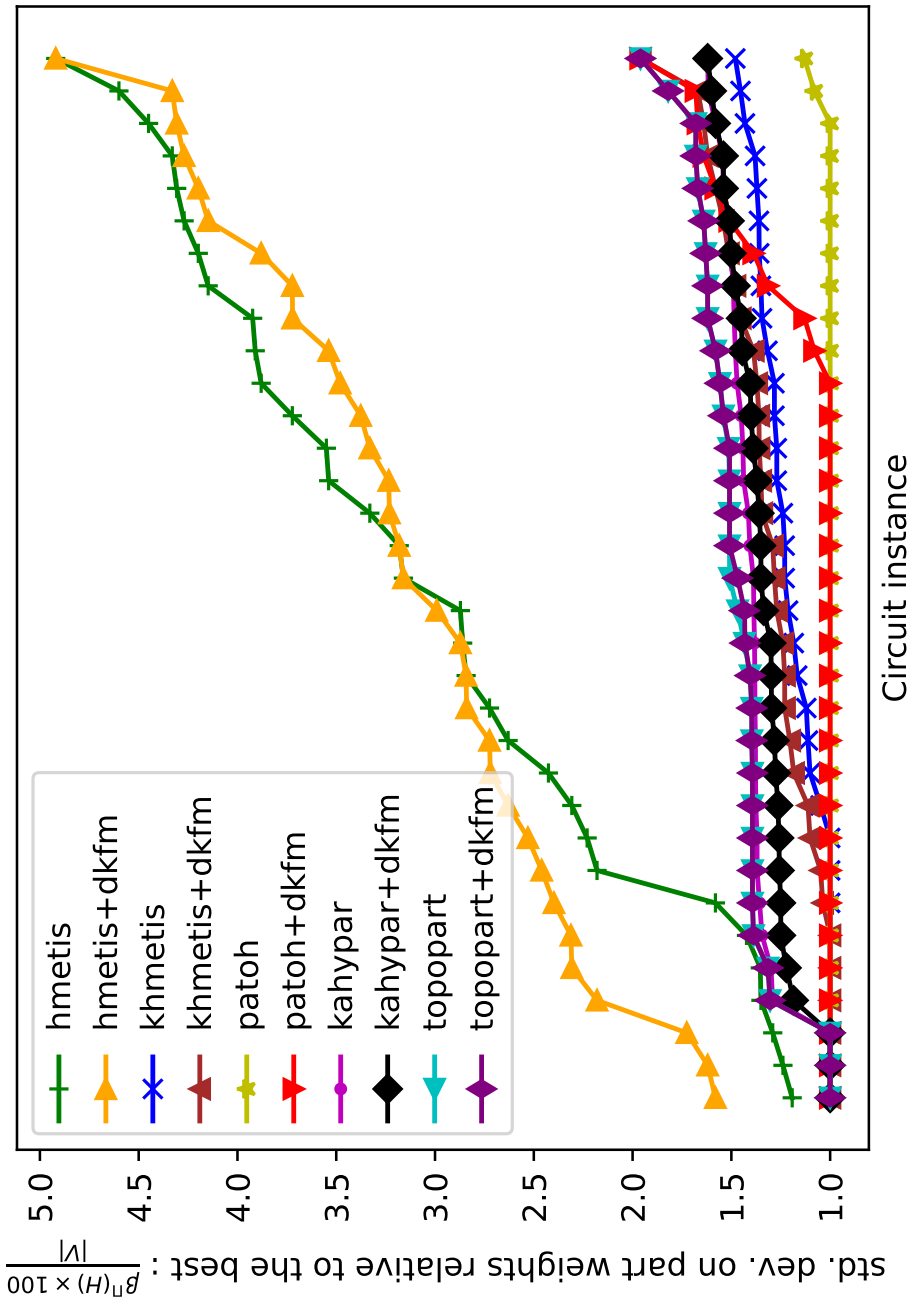


Figure A.8: Balance cost relative to the best on target topology T5 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFM.

Table A.57: Results for balance cost: $\beta(H^{\Pi})$, on target T1 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	4.92	4.92	1.00	1.00	1.00	1.00	1.00	1.00	2.96	2.96
b02	4.33	4.33	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
b03	1.00	2.23	1.00	2.23	1.00	2.23	1.62	1.62	2.23	2.23
b04	1.93	2.33	1.53	2.33	1.00	1.40	2.01	1.80	2.07	2.07
b05	4.30	4.30	1.00	1.10	1.19	1.19	1.79	1.10	1.97	2.16
b06	6.17	6.17	1.00	1.00	1.00	1.00	1.00	1.00	2.72	2.72
b07	6.18	6.18	1.68	1.23	1.23	1.23	2.04	1.68	2.35	2.35
b08	3.16	3.16	1.54	1.54	1.00	1.00	2.08	2.08	2.08	2.08
b09	5.65	4.49	1.58	2.74	1.00	1.00	2.04	1.58	2.74	2.74
b10	4.37	4.37	1.00	1.96	1.48	1.48	1.58	1.96	1.96	2.44
b11	5.09	2.92	1.38	2.28	1.26	2.28	1.97	2.28	2.28	2.28
b12	4.71	4.71	1.93	1.83	1.00	1.00	2.07	1.93	2.11	2.21
b13	4.54	4.54	1.82	1.54	1.00	1.00	1.51	1.27	2.36	2.36
b14	5.56	5.56	2.15	2.14	1.26	1.26	2.08	1.65	2.23	2.25
b17	4.13	4.13	2.14	2.09	1.49	1.49	2.22	2.06	1.82	2.25
b18	1.96	2.32	1.97	1.97	1.50	1.50	2.15	2.12	2.20	2.20
b19	1.98	2.42	1.16	1.16	1.26	1.26	2.19	1.83	1.11	1.11
b20	2.13	5.75	1.67	1.67	1.42	2.25	2.20	1.87	2.18	2.25
b21	1.68	5.35	1.13	1.13	1.31	2.25	1.98	1.41	2.23	2.25
b22	1.64	6.03	2.18	2.17	1.35	1.35	2.18	1.82	1.94	2.25

Table A.58: Results for balance cost: $\beta(H^{\Pi})$, on target T2 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
B01	4.92	4.92	1.00	2.96	1.00	1.00	1.00	1.00	2.96	2.96
B02	4.33	4.33	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
B03	1.00	2.23	1.00	2.23	1.00	1.00	1.62	2.23	2.23	2.23
B04	1.93	1.93	1.53	1.00	1.00	2.33	2.01	1.93	2.33	2.33
B05	4.30	4.30	1.00	1.10	1.19	1.19	1.79	1.10	2.26	2.26
B06	6.17	4.45	1.00	1.00	1.00	1.00	1.00	1.00	2.72	2.72
B07	6.18	6.18	1.68	2.35	1.23	1.23	2.04	1.68	2.35	2.35
B08	3.16	3.16	1.54	1.54	1.00	2.08	2.08	1.54	2.08	2.08
B09	5.65	4.49	1.58	2.74	1.00	1.00	2.04	1.58	2.74	2.74
B10	4.37	4.37	1.00	2.44	1.48	2.44	1.58	1.48	2.44	2.44
B11	5.09	3.94	1.38	2.28	1.26	2.28	1.97	1.51	2.28	2.15
B12	4.71	4.71	1.93	1.83	1.00	1.09	2.07	1.93	2.21	2.21
B13	4.54	4.27	1.82	1.54	1.00	1.00	1.51	1.27	2.36	2.36
B14	5.56	5.56	2.15	2.24	1.26	1.26	2.08	1.65	2.25	2.25
B17	4.13	4.13	2.14	2.09	1.49	1.49	2.22	1.95	2.25	2.25
B18	1.96	2.32	1.97	1.97	1.50	1.50	2.15	2.12	1.89	1.89
B19	1.98	2.42	1.16	1.16	1.26	1.26	2.19	1.83	2.25	2.25
B20	2.13	5.75	1.67	1.67	1.42	2.25	2.20	1.87	2.25	2.24
B21	1.68	5.35	1.13	1.13	1.31	2.25	1.98	1.41	2.25	2.00
B22	1.64	6.03	2.18	2.18	1.35	1.35	2.18	2.05	2.25	2.10

Table A.59: Results for balance cost: $\beta(H^{\Pi})$, on target T3 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	4.92	4.92	1.00	2.96	1.00	1.00	1.00	2.96	2.96	2.96
b02	4.33	4.33	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
b03	1.00	1.00	1.00	2.23	1.00	1.00	1.62	1.62	2.23	2.23
b04	1.93	1.93	1.53	1.00	1.00	2.33	2.01	2.07	2.20	2.20
b05	4.30	4.30	1.00	1.10	1.19	1.19	1.79	1.10	2.26	2.26
b06	6.17	4.45	1.00	1.00	1.00	1.00	1.00	1.00	2.72	2.72
b07	6.18	6.18	1.68	1.23	1.23	1.23	2.04	1.68	2.35	2.35
b08	3.16	3.16	1.54	1.54	1.00	2.08	2.08	2.08	1.54	1.54
b09	5.65	4.49	1.58	2.74	1.00	1.00	2.04	1.58	2.74	2.74
b10	4.37	3.88	1.00	1.96	1.48	1.48	1.58	1.48	2.44	2.44
b11	5.09	2.53	1.38	2.28	1.26	2.28	1.97	1.51	2.28	2.28
b12	4.71	4.71	1.93	2.21	1.00	1.00	2.07	1.93	2.21	2.21
b13	4.54	4.54	1.82	2.36	1.00	1.82	1.51	1.27	2.09	2.09
b14	5.56	5.39	2.15	2.14	1.26	1.26	2.08	1.65	2.25	2.25
b17	4.13	4.13	2.14	2.09	1.49	1.49	2.22	2.06	2.25	2.25
b18	1.96	2.32	1.97	1.97	1.50	1.50	2.15	2.12	2.25	2.25
b19	1.98	2.42	1.16	1.16	1.26	1.26	2.19	1.83	2.13	2.13
b20	2.13	5.75	1.67	1.67	1.42	2.24	2.20	1.52	2.22	2.25
b21	1.68	5.35	1.13	1.13	1.31	2.25	1.98	1.38	2.25	2.25
b22	1.64	6.03	2.18	2.18	1.35	1.35	2.18	2.05	2.25	2.25

Table A.60: Results for balance cost: $\beta(H^{\Pi})$, on target T4 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
B01	4.92	4.92	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
B02	4.33	4.33	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
B03	2.23	1.62	1.00	1.62	1.00	1.62	1.62	1.62	1.62	1.62
B04	2.86	2.73	1.27	1.67	1.00	1.00	1.50	1.40	1.67	1.67
B05	3.91	3.91	1.00	1.00	1.00	1.68	1.51	1.39	1.68	1.68
B06	4.45	4.45	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
B07	4.60	3.03	1.23	1.23	1.00	1.68	1.45	1.45	1.68	1.68
B08	3.16	3.16	1.00	1.54	1.00	1.00	1.38	1.54	1.54	1.54
B09	3.33	2.74	1.00	1.58	1.00	1.58	1.58	1.58	1.58	1.58
B10	3.88	3.88	1.48	1.96	1.00	1.00	1.48	1.48	1.96	1.96
B11	3.55	3.55	1.38	1.64	1.00	1.00	1.47	1.51	1.64	1.64
B12	3.13	2.76	1.56	1.65	1.09	1.09	1.54	1.37	1.65	1.65
B13	4.27	4.27	1.27	1.27	1.00	1.82	1.54	1.54	1.82	1.82
B14	4.08	3.48	1.37	1.62	1.04	1.62	1.61	1.56	1.62	1.62
B17	1.51	3.95	1.50	1.51	1.17	1.17	1.62	1.59	1.63	1.63
B18	1.49	2.89	1.48	1.48	1.25	1.25	1.61	1.58	1.63	1.63
B19	1.40	1.95	1.39	1.39	1.13	1.13	1.56	1.38	1.63	1.63
B20	1.58	3.78	1.58	1.58	1.17	1.63	1.59	1.50	1.63	1.63
B21	1.58	3.59	1.51	1.51	1.11	1.63	1.60	1.56	1.63	1.63
B22	1.57	3.23	1.59	1.63	1.16	1.16	1.59	1.45	1.63	1.63

Table A.61: Results for balance cost: $\beta(H^{\Pi})$, on target T5 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	4.92	4.92	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
b02	4.33	4.33	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
b03	2.23	1.62	1.00	1.62	1.00	1.00	1.62	1.62	1.62	1.62
b04	2.86	2.46	1.27	1.67	1.00	1.67	1.50	1.40	1.67	1.67
b05	3.91	2.84	1.00	1.00	1.00	1.00	1.51	1.39	1.68	1.68
b06	4.45	2.72	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
b07	4.60	3.48	1.23	1.23	1.00	1.68	1.45	1.45	1.68	1.68
b08	3.16	3.16	1.00	1.54	1.00	1.00	1.38	1.54	1.54	1.54
b09	3.33	3.33	1.00	1.58	1.00	1.58	1.58	1.58	1.58	1.58
b10	3.88	3.88	1.48	1.96	1.00	1.96	1.48	1.48	1.96	1.96
b11	3.55	2.53	1.38	1.64	1.00	1.64	1.47	1.51	1.64	1.64
b12	3.13	3.13	1.56	1.65	1.09	1.09	1.54	1.37	1.65	1.56
b13	4.27	4.27	1.27	1.27	1.00	1.00	1.54	1.54	1.82	1.82
b14	4.08	3.87	1.37	1.54	1.04	1.04	1.61	1.56	1.62	1.62
b17	1.51	3.95	1.50	1.51	1.17	1.17	1.62	1.59	1.63	1.63
b18	1.49	2.89	1.48	1.48	1.25	1.25	1.61	1.58	1.63	1.63
b19	1.40	1.95	1.39	1.39	1.13	1.13	1.56	1.38	1.62	1.62
b20	1.58	3.78	1.58	1.58	1.17	1.17	1.59	1.52	1.63	1.63
b21	1.58	3.59	1.51	1.51	1.11	1.11	1.60	1.56	1.63	1.63
b22	1.57	3.47	1.59	1.59	1.16	1.16	1.59	1.45	1.63	1.63

Table A.62: Results for balance cost: $\beta(H^{\Pi})$, on target T6 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
B01	4.92	4.92	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
B02	4.33	4.33	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
B03	2.23	1.62	1.00	1.00	1.00	1.62	1.62	1.62	1.62	1.62
B04	2.86	2.73	1.27	1.67	1.00	1.00	1.50	1.40	1.67	1.67
B05	3.91	2.94	1.00	1.00	1.00	1.00	1.51	1.39	1.68	1.68
B06	4.45	2.72	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
B07	4.60	3.48	1.23	1.23	1.00	1.68	1.45	1.45	1.68	1.68
B08	3.16	3.16	1.00	1.00	1.00	1.00	1.38	1.54	1.54	1.54
B09	3.33	3.33	1.00	1.00	1.00	1.00	1.58	1.00	1.58	1.58
B10	3.88	3.88	1.48	1.48	1.00	1.00	1.48	1.48	1.96	1.96
B11	3.55	3.55	1.38	1.38	1.00	1.51	1.47	1.51	1.64	1.64
B12	3.13	2.67	1.56	1.56	1.09	1.09	1.54	1.37	1.65	1.56
B13	4.27	4.27	1.27	1.27	1.00	1.82	1.54	1.54	1.82	1.82
B14	4.08	4.07	1.37	1.37	1.04	1.62	1.61	1.61	1.61	1.62
B17	1.51	1.63	1.50	1.63	1.17	1.17	1.62	1.59	1.63	1.63
B18	1.49	1.49	1.48	1.48	1.25	1.25	1.61	1.58	1.62	1.62
B19	1.40	1.40	1.39	1.39	1.13	1.13	1.56	1.38	1.62	1.62
B20	1.58	1.63	1.58	1.58	1.17	1.17	1.59	1.52	1.63	1.63
B21	1.58	1.58	1.51	1.51	1.11	1.11	1.60	1.56	1.62	1.62
B22	1.57	1.62	1.59	1.59	1.16	1.16	1.59	1.45	1.63	1.63

Table A.63: Results for balance cost: $\beta(H^{\Pi})$, on target T1 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	5.07	5.04	2.14	2.25	1.25	1.25	2.23	2.10	2.09	2.09
mnist	3.15	3.07	1.00	1.39	1.49	2.26	1.77	1.52	2.26	2.26
mobilnet1	3.66	3.66	1.24	1.24	1.25	1.25	2.19	1.73	2.25	2.25
OneCore	5.98	5.98	2.11	2.11	1.00	2.25	2.19	1.95	1.93	2.25
PuLSAR	4.91	4.91	1.84	1.84	1.30	1.30	2.13	1.80	2.25	2.25
WasgaServer	5.62	5.62	2.18	2.18	1.50	1.50	2.06	1.65	2.24	2.24
bitonic_mesh	3.38	3.38	1.00	1.00	1.25	1.25	1.74	1.35	2.14	2.14
cholesky_bdti	4.01	3.85	1.22	1.22	1.00	1.00	1.91	1.54	2.11	2.11
dart	1.81	1.81	1.00	1.00	1.25	1.25	2.21	2.11	2.25	2.25
denoise	2.95	2.95	1.73	1.73	1.25	1.25	2.17	2.05	2.24	2.24
des90	3.50	3.35	1.00	1.00	1.00	1.00	1.89	1.29	2.25	2.25
xge_mac	5.07	5.07	1.52	1.52	1.43	1.43	2.25	2.25	2.25	2.25
cholesky_mc	4.58	4.58	1.19	1.34	1.25	1.25	2.12	1.96	1.90	2.25

Table A.64: Results for balance cost: $\beta(H^{\Pi})$, on target T2 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	5.07	4.88	2.14	2.14	1.25	1.25	2.23	2.10	2.25	2.25
mnist	3.15	3.15	1.00	1.39	1.49	2.25	1.77	1.52	2.26	2.26
mobilnet1	3.66	3.66	1.24	1.24	1.25	1.25	2.19	1.93	2.25	2.25
OneCore	5.98	5.98	2.11	2.25	1.00	1.93	2.19	1.27	2.25	2.25
PuLSAR	4.91	4.91	1.84	1.84	1.30	1.30	2.13	1.80	2.25	2.25
WasgaServer	5.62	5.62	2.18	2.18	1.50	1.50	2.06	1.65	2.25	2.25
bitonic_mesh	3.38	3.21	1.00	1.00	1.25	1.25	1.74	1.35	2.22	2.22
cholesky_bdti	4.01	3.93	1.22	1.22	1.00	1.00	1.91	1.54	2.25	2.25
dart	1.81	1.81	1.00	1.00	1.25	1.25	2.21	2.11	2.25	2.25
denoise	2.95	2.95	1.73	1.73	1.25	1.25	2.17	2.05	2.25	2.25
des90	3.50	3.48	1.00	1.00	1.00	1.00	1.89	1.29	2.25	2.25
xge_mac	5.07	5.07	1.52	1.52	1.43	2.23	2.25	2.25	2.25	2.23
cholesky_mc	4.58	4.58	1.19	1.19	1.25	1.81	2.12	1.99	2.07	2.07

Table A.65: Results for balance cost: $\beta(H^{\Pi})$, on target T3 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	5.07	5.04	2.14	2.14	1.25	1.25	2.23	2.11	1.45	1.45
mnist	3.15	3.07	1.00	1.00	1.49	2.26	1.77	1.70	2.26	2.26
mobilnet1	3.66	3.66	1.24	1.24	1.25	1.25	2.19	1.78	2.25	2.25
OneCore	5.98	5.98	2.11	2.11	1.00	1.93	2.19	1.95	2.25	2.25
PuLSAR	4.91	4.91	1.84	1.84	1.30	1.30	2.13	1.80	2.25	2.25
WasgaServer	5.62	5.62	2.18	2.18	1.50	1.50	2.06	1.65	2.24	2.24
bitonic_mesh	3.38	3.38	1.00	1.00	1.25	1.25	1.74	1.35	2.25	2.25
cholesky_bdti	4.01	4.01	1.22	1.22	1.00	1.00	1.91	1.54	1.55	1.55
dart	1.81	1.81	1.00	1.00	1.25	1.25	2.21	2.11	2.25	2.25
denoise	2.95	2.95	1.73	1.73	1.25	1.25	2.17	2.05	1.55	1.55
des90	3.50	3.50	1.00	1.00	1.00	1.00	1.89	1.29	2.25	2.25
xge_mac	5.07	5.07	1.52	1.52	1.43	1.43	2.25	2.25	2.25	2.25
cholesky_mc	4.58	4.58	1.19	1.19	1.25	1.25	2.12	1.99	1.87	1.87

Table A.66: Results for balance cost: $\beta(H^{\Pi})$, on target T4 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	4.48	4.48	1.57	1.57	1.08	1.08	1.61	1.56	1.63	1.63
mnist	1.96	1.96	1.30	1.30	1.24	1.63	1.61	1.47	1.63	1.63
mobilnet1	3.69	3.69	1.29	1.29	1.16	1.16	1.59	1.55	1.62	1.62
OneCore	4.87	4.87	1.56	1.56	1.16	1.62	1.61	1.55	1.62	1.62
PuLSAR	5.04	5.04	1.45	1.45	1.17	1.17	1.62	1.58	1.63	1.63
WasgaServer	4.14	4.14	1.59	1.59	1.17	1.17	1.57	1.50	1.63	1.63
bitonic_mesh	2.18	2.18	1.00	1.00	1.13	1.13	1.49	1.25	1.62	1.62
cholesky_bdti	2.62	2.62	1.31	1.31	1.08	1.08	1.55	1.47	1.63	1.63
dart	2.63	2.63	1.00	1.00	1.08	1.08	1.62	1.60	1.63	1.63
denoise	2.70	2.70	1.50	1.50	1.17	1.17	1.61	1.58	1.63	1.63
des90	3.07	3.07	1.19	1.19	1.08	1.08	1.52	1.28	1.63	1.63
xge_mac	3.16	3.16	1.30	1.30	1.16	1.61	1.61	1.59	1.61	1.61
cholesky_mc	4.02	4.02	1.26	1.12	1.08	1.08	1.49	1.42	1.63	1.63

Table A.67: Results for balance cost: $\beta(H^{\Pi})$, on target T5 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	4.48	4.48	1.57	1.57	1.08	1.08	1.61	1.56	1.63	1.63
mnist	1.96	1.96	1.30	1.30	1.24	1.63	1.61	1.61	1.63	1.63
mobilnet1	3.69	3.69	1.29	1.29	1.16	1.16	1.59	1.48	1.62	1.62
OneCore	4.87	4.87	1.56	1.56	1.16	1.16	1.61	1.55	1.62	1.62
PuLSAR	5.04	5.04	1.45	1.45	1.17	1.17	1.62	1.58	1.63	1.63
WasgaServer	4.14	4.14	1.59	1.59	1.17	1.17	1.57	1.50	1.63	1.63
bitonic_mesh	2.18	2.18	1.00	1.00	1.13	1.13	1.49	1.25	1.62	1.62
cholesky_bdti	2.62	2.59	1.31	1.30	1.08	1.08	1.55	1.40	1.63	1.63
dart	2.63	2.63	1.00	1.00	1.08	1.08	1.62	1.60	1.63	1.63
denoise	2.70	2.70	1.50	1.50	1.17	1.17	1.61	1.58	1.63	1.63
des90	3.07	3.07	1.19	1.19	1.08	1.63	1.52	1.28	1.63	1.63
xge_mac	3.16	3.16	1.30	1.61	1.16	1.61	1.61	1.59	1.61	1.61
cholesky_mc	4.02	4.02	1.26	1.12	1.08	1.08	1.49	1.36	1.63	1.63

Table A.68: Results for balance cost: $\beta(H^{\Pi})$, on target T6 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	4.48	4.48	1.57	1.57	1.08	1.08	1.61	1.56	1.61	1.61
mnist	1.96	1.96	1.30	1.63	1.24	1.30	1.61	1.56	1.63	1.63
mobilnet1	3.69	3.69	1.29	1.29	1.16	1.16	1.59	1.48	1.62	1.62
OneCore	4.87	4.87	1.56	1.56	1.16	1.16	1.61	1.59	1.62	1.62
PuLSAR	5.04	5.04	1.45	1.45	1.17	1.17	1.62	1.58	1.63	1.63
WasgaServer	4.14	4.14	1.59	1.59	1.17	1.17	1.57	1.50	1.62	1.62
bitonic_mesh	2.18	1.96	1.00	1.00	1.13	1.13	1.49	1.25	1.62	1.62
cholesky_bdti	2.62	2.62	1.31	1.31	1.08	1.08	1.55	1.40	1.60	1.60
dart	2.63	2.63	1.00	1.00	1.08	1.08	1.62	1.60	1.62	1.62
denoise	2.70	2.70	1.50	1.50	1.17	1.17	1.61	1.58	1.63	1.63
des90	3.07	3.07	1.19	1.19	1.08	1.08	1.52	1.28	1.62	1.62
xge_mac	3.16	3.16	1.30	1.61	1.16	1.16	1.61	1.59	1.61	1.61
cholesky_mc	4.02	4.02	1.26	1.16	1.08	1.08	1.49	1.36	1.61	1.61

A.3.2 Numerical results of DKFMFAST

Critical path results

Results on critical path degradation presented in Chapter 6 are based on tables introduced in this subsection. Each table shows us effect of DKFMFAST on critical path degradation of partition produced by min-cut algorithms.

Table A.69: Results of DKFMFAST effects on critical path: $d_{\max}^{\Pi}(H)$, on target T1 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
B01	7.87	7.87	7.87	7.57	10.31	10.31	8.18	7.72	13.05	13.05
B02	12.31	12.31	12.31	12.13	9.09	9.09	15.18	15.18	12.31	12.31
B03	6.06	6.06	4.63	8.06	7.40	5.88	7.61	6.16	13.01	13.01
B04	4.03	4.03	3.35	3.53	3.35	3.35	3.36	2.37	11.42	11.42
B05	0.93	0.93	0.64	0.96	1.18	1.18	1.15	0.74	9.07	9.07
B06	11.96	11.96	12.31	12.13	12.31	12.31	14.98	12.31	15.36	15.36
B07	2.94	2.94	3.61	2.78	3.04	3.04	3.17	2.55	7.07	7.07
B08	4.98	4.98	5.22	5.22	5.83	5.83	5.11	4.98	7.29	7.29
B09	16.14	16.14	10.78	7.21	9.00	9.00	11.50	9.00	10.78	10.78
B10	8.23	8.23	6.78	6.78	8.23	8.23	7.76	6.86	9.59	9.59
B11	3.36	3.36	2.81	2.81	4.01	4.01	3.54	2.34	7.48	7.48
B12	4.16	4.16	3.24	3.76	3.24	3.24	3.98	3.24	10.10	10.10
B13	1.71	1.71	1.27	2.10	2.89	2.89	2.00	1.95	9.21	9.21
B14	2.12	2.12	2.11	2.42	2.39	2.39	2.17	1.56	11.35	11.35
B17	0.68	0.68	0.48	1.65	0.81	0.81	0.65	0.37	13.49	13.49
B18	0.10	0.21	0.21	0.21	0.11	0.11	0.21	0.21	8.15	8.15
B19	0.30	0.28	0.18	0.18	0.31	0.31	0.29	0.28	8.59	8.59
B20	4.01	1.52	2.17	2.17	2.24	2.24	1.98	1.03	11.73	11.73
B21	2.99	0.98	2.00	2.00	2.21	2.21	1.76	1.03	10.03	10.03
B22	1.63	1.22	1.78	1.78	0.88	0.88	1.52	0.47	12.27	12.26

Table A.70: Results of DKFMFAST effects on critical path: $d^{\Pi}_{\max}(H)$, on target T2 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	10.31	10.31	13.05	17.63	12.75	12.75	10.59	10.01	20.37	20.37
b02	15.18	15.18	18.05	15.36	18.41	12.31	17.01	15.18	24.33	24.33
b03	7.87	7.87	7.87	9.68	7.68	7.68	9.56	9.30	12.72	12.72
b04	4.84	4.84	4.07	3.99	5.76	5.76	4.46	2.65	9.61	9.61
b05	1.22	1.22	1.27	0.96	1.22	1.22	1.50	1.17	5.62	5.62
b06	21.46	21.46	15.18	12.13	15.18	15.18	19.22	18.23	27.20	27.20
b07	3.00	3.00	3.83	3.11	3.04	3.04	3.75	2.74	12.33	11.27
b08	5.22	5.22	7.17	7.17	6.69	6.69	6.33	4.98	15.51	15.51
b09	17.93	17.93	17.93	7.21	10.78	10.78	15.07	12.57	21.50	21.50
b10	8.23	8.23	7.91	8.15	10.32	8.96	8.49	7.91	16.01	16.01
b11	4.74	4.74	4.89	4.83	6.99	6.99	4.91	3.36	7.99	7.99
b12	4.16	4.16	3.24	3.76	3.61	3.61	4.94	4.11	11.44	11.44
b13	3.38	3.38	1.81	2.10	2.89	2.89	2.17	1.95	8.38	8.38
b14	2.12	2.12	3.23	2.98	3.94	3.94	2.80	1.98	10.13	10.13
b17	0.68	0.68	0.81	1.65	0.81	0.81	0.67	0.41	10.32	9.76
b18	0.20	0.21	0.21	0.21	0.31	0.31	0.29	0.21	6.44	6.44
b19	0.30	0.38	0.38	0.38	0.31	0.31	0.34	0.28	7.04	7.04
b20	4.01	1.52	2.17	2.17	3.26	3.26	2.29	1.03	11.33	11.33
b21	2.99	0.98	2.00	2.00	2.21	2.21	2.59	1.73	9.59	9.32
b22	1.63	1.77	3.80	3.80	0.88	0.88	1.78	0.47	8.99	8.99

Table A.71: Results of DKFMFAST effects on critical path: $d^{\Pi}_{\max}(H)$, on target T3 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	5.28	5.28	7.87	7.42	7.72	7.72	7.42	7.42	10.31	10.31
b02	9.26	9.26	9.26	9.26	6.21	6.21	12.13	12.13	12.31	12.31
b03	4.54	4.54	4.63	6.44	4.45	4.45	6.97	6.16	11.39	11.39
b04	3.03	3.03	2.15	2.79	3.03	3.03	2.69	2.05	4.23	4.23
b05	0.93	0.93	0.64	0.64	1.18	1.18	1.06	0.74	5.91	5.82
b06	9.26	9.26	6.21	6.21	6.21	6.21	11.56	9.09	12.31	12.31
b07	2.40	2.40	2.52	2.21	2.74	2.74	2.44	2.01	5.85	5.34
b08	2.90	2.90	4.18	4.18	3.58	3.58	3.82	2.90	7.29	7.29
b09	10.78	10.78	9.00	5.42	7.21	7.21	9.17	9.00	10.78	10.78
b10	6.86	6.86	5.42	5.42	6.86	6.86	6.31	5.19	7.99	7.99
b11	1.76	1.76	1.93	1.93	3.01	3.01	2.21	1.84	3.18	3.18
b12	3.29	3.29	2.36	2.36	2.16	1.79	3.22	2.36	6.80	5.92
b13	0.88	0.88	1.27	2.10	2.05	2.05	2.00	1.95	3.37	3.37
b14	1.84	1.84	1.57	1.99	1.80	1.80	1.49	1.15	4.63	4.63
b17	0.68	0.68	0.48	0.75	0.81	0.81	0.65	0.37	5.56	5.56
b18	0.10	0.11	0.10	0.10	0.11	0.11	0.12	0.10	3.52	3.52
b19	0.20	0.18	0.18	0.18	0.31	0.31	0.21	0.20	4.02	4.01
b20	2.48	1.11	2.17	2.17	1.99	1.99	1.41	1.02	6.02	6.01
b21	1.74	0.98	2.00	2.00	1.96	1.96	1.53	1.03	5.65	5.42
b22	1.12	0.78	0.86	0.86	0.70	0.70	0.85	0.45	5.93	5.70

Table A.72: Results of DKFMFAST effects on critical path: $d^{\Pi}_{\max}(H)$, on target T4 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	15.49	15.49	15.49	15.64	17.93	17.93	20.86	15.49	30.74	30.74
b02	18.41	18.41	21.28	27.55	24.51	24.51	31.09	24.33	21.28	21.28
b03	12.54	12.54	21.10	14.53	21.10	21.10	17.81	11.01	20.91	20.91
b04	7.84	7.84	8.98	7.43	7.39	7.39	7.28	5.94	9.65	9.65
b05	3.67	3.67	2.88	3.48	3.69	3.69	4.23	2.80	5.58	5.58
b06	18.41	18.41	24.33	30.25	21.46	21.46	30.48	24.33	42.80	42.80
b07	7.46	7.46	7.10	8.16	7.87	6.79	7.66	6.01	11.33	11.33
b08	8.08	8.08	15.45	18.61	15.45	15.45	13.85	8.32	14.47	14.47
b09	23.28	23.28	14.35	19.71	23.28	21.50	20.07	14.35	25.07	25.07
b10	21.69	21.69	16.16	17.53	13.68	13.68	19.49	13.68	23.05	23.05
b11	6.23	6.23	4.89	8.10	4.42	4.42	7.00	5.35	7.84	7.84
b12	9.38	9.38	6.80	8.45	15.42	15.42	8.22	6.44	14.49	13.61
b13	1.27	1.27	2.84	2.64	6.87	6.87	7.26	4.60	7.89	7.89
b14	5.53	5.14	5.76	4.83	5.84	5.84	4.90	3.78	11.57	11.57
b17	2.70	2.70	3.81	2.43	3.85	3.85	2.47	1.50	7.75	7.41
b18	0.41	0.35	0.71	0.71	0.93	0.93	0.43	0.25	4.93	4.93
b19	0.75	0.79	0.58	0.58	0.38	0.38	0.66	0.51	5.61	5.47
b20	2.88	2.73	6.05	6.05	5.11	5.11	4.72	3.33	9.80	9.80
b21	6.10	5.02	4.50	4.50	4.71	4.71	4.33	3.27	8.86	8.43
b22	4.02	2.90	4.73	4.73	3.34	3.34	3.73	2.69	7.84	7.84

Table A.73: Results of DKFMFAST effects on critical path: $d^{\Pi}_{\max}(H)$, on target T5 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
B01	20.52	20.52	23.11	20.67	20.52	20.52	25.61	20.67	38.81	38.81
B02	18.41	18.41	30.60	24.33	27.55	27.55	32.68	24.51	30.43	30.43
B03	15.77	15.77	22.72	11.20	22.72	22.72	16.57	14.53	32.24	30.71
B04	12.06	12.06	9.61	10.04	9.30	9.30	9.13	7.32	9.44	9.30
B05	4.34	4.34	3.38	4.74	3.69	3.69	4.90	3.43	8.42	8.42
B06	36.70	36.70	30.43	27.38	21.46	21.46	35.34	24.51	21.28	21.28
B07	6.94	6.94	10.88	10.34	8.93	8.93	9.25	7.42	14.99	14.99
B08	15.57	15.57	16.48	19.59	15.45	15.45	19.50	15.45	19.71	19.71
B09	26.85	26.85	19.71	21.50	30.43	30.43	22.57	17.51	25.07	25.07
B10	21.69	21.69	19.05	14.80	17.76	17.76	21.09	17.68	16.24	16.16
B11	6.73	6.73	8.78	6.79	6.20	6.20	7.83	5.91	12.95	12.95
B12	12.01	12.01	11.18	12.84	17.12	17.12	10.73	9.07	10.15	10.15
B13	2.93	2.93	5.15	5.05	6.87	6.87	7.21	4.60	6.66	6.66
B14	5.69	5.69	6.45	5.00	7.69	7.69	6.68	4.81	9.31	9.31
B17	1.51	2.29	2.60	2.74	4.21	4.21	2.79	1.54	7.07	7.07
B18	0.38	0.42	0.98	0.98	1.32	1.32	0.66	0.42	5.52	5.11
B19	0.89	1.00	0.96	0.96	0.68	0.68	0.81	0.69	4.04	4.04
B20	5.59	5.01	5.98	5.98	6.25	6.25	5.45	4.24	8.45	8.45
B21	4.27	5.00	4.46	4.46	5.71	5.71	4.63	3.59	11.11	11.11
B22	4.49	4.36	5.05	5.05	4.72	4.72	4.79	3.47	7.34	7.34

Table A.74: Results of DKFMFAST effects on critical path: $d^{\Pi}_{\max}(H)$, on target T6 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	7.87	7.87	7.87	7.87	7.72	7.72	9.93	7.72	15.64	15.64
b02	9.26	9.26	12.31	12.31	9.26	9.26	13.17	12.13	15.36	15.36
b03	4.92	4.92	8.15	8.15	8.15	8.15	8.10	6.53	11.29	11.29
b04	4.23	4.23	4.23	4.23	3.35	3.35	3.58	2.72	8.98	8.94
b05	1.81	1.81	1.80	1.80	1.83	1.83	1.82	1.22	5.77	5.47
b06	12.31	12.31	9.26	9.26	9.26	9.26	13.76	12.13	12.13	12.13
b07	4.19	4.19	4.89	4.89	4.06	4.06	4.06	3.29	7.65	7.65
b08	6.25	6.25	8.26	8.26	6.13	6.13	8.12	7.17	10.39	10.39
b09	10.78	10.78	9.00	9.00	10.78	10.78	10.78	9.00	14.35	14.35
b10	6.86	6.86	6.78	6.78	6.86	6.86	8.44	7.75	9.51	9.51
b11	1.93	1.93	2.34	2.34	2.34	2.34	3.01	2.43	4.47	4.47
b12	4.99	4.99	4.11	4.11	4.94	4.94	4.25	3.39	7.42	6.75
b13	1.27	1.27	1.70	1.70	2.05	2.05	2.93	2.34	7.35	7.35
b14	2.00	2.00	2.37	2.37	3.14	3.14	2.54	1.84	6.54	6.54
b17	0.86	0.86	1.17	1.17	1.45	1.45	0.98	0.86	5.02	4.91
b18	0.20	0.20	0.21	0.21	0.33	0.33	0.26	0.21	4.16	4.16
b19	0.28	0.28	0.28	0.28	0.21	0.21	0.29	0.26	4.86	4.86
b20	2.02	2.02	2.48	2.48	2.48	2.48	1.90	1.54	6.79	6.77
b21	2.08	2.08	1.70	1.70	2.23	2.23	1.87	1.40	5.91	5.91
b22	2.08	2.08	1.83	1.83	1.45	1.45	1.59	1.33	6.23	6.23

Table A.75: Results of DKFMFAST effects on critical path: $d^{\Pi}_{\max}(H)$, on target T1 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	kHMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	0.62	0.62	1.66	1.66	3.91	3.91	0.49	0.03	13.06	12.92
mnist	9.02	5.69	5.97	5.97	10.83	10.83	4.97	2.45	17.86	17.86
mobilnet1	1.82	1.82	1.72	1.72	6.78	6.78	1.83	1.10	10.08	10.08
OneCore	1.25	1.25	0.11	0.11	2.40	2.40	0.47	0.01	10.37	9.95
PuLSAR	0.49	0.49	0.09	0.09	3.82	3.82	0.39	0.08	11.30	11.30
WasgaServer	0.03	0.03	0.67	0.67	4.92	4.92	0.38	0.01	13.19	13.19
bitonic_mesh	0.71	0.71	1.42	1.42	12.70	12.70	0.71	0.71	12.07	12.07
cholesky_bdti	1.90	1.90	1.90	1.90	4.83	4.83	2.10	1.13	10.96	10.96
dart	1.97	1.97	1.69	1.69	2.79	2.79	2.75	1.41	6.71	6.71
denoise	0.00	0.00	0.00	0.00	0.07	0.07	0.00	0.00	6.85	6.85
des90	2.94	2.94	1.42	1.42	11.27	11.27	0.95	0.58	12.21	12.21
xge_mac	2.50	2.50	3.98	3.98	6.77	6.77	4.28	3.98	8.67	8.59
cholesky_mc	1.08	1.08	2.00	2.00	4.88	4.88	1.67	0.97	12.87	12.87

Table A.76: Results of DKFMFAST effects on critical path: $d^{\Pi}_{\max}(H)$, on target T2 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	0.84	0.84	2.08	2.08	3.92	3.92	0.80	0.36	20.92	20.92
mnist	9.02	8.93	5.97	5.97	17.49	17.49	5.16	2.73	30.71	30.71
mobilnet1	1.82	1.82	2.82	2.82	7.99	7.99	1.83	1.10	10.01	10.01
OneCore	1.25	1.25	1.72	1.72	4.25	4.20	0.74	0.01	17.67	17.46
PuLSAR	0.49	0.49	1.12	0.95	7.13	7.13	0.77	0.13	19.04	18.63
WasgaServer	0.44	0.44	2.26	2.26	5.74	5.74	0.56	0.01	23.91	23.91
bitonic_mesh	1.34	1.34	4.46	4.46	18.66	18.66	1.43	1.34	18.83	18.83
cholesky_bdti	4.67	4.67	2.00	2.00	6.68	6.68	2.49	1.90	19.89	19.89
dart	1.97	1.97	2.25	2.25	2.80	2.80	3.29	1.77	8.68	8.68
denoise	0.00	0.00	0.00	0.00	0.07	0.07	0.00	0.00	7.13	7.13
des90	2.94	2.94	1.47	1.47	13.20	13.20	1.62	1.34	18.97	18.97
xge_mac	2.50	2.50	5.12	5.12	9.90	9.90	4.28	3.98	20.59	20.59
cholesky_mc	2.11	2.11	2.00	2.00	5.65	5.65	2.87	1.90	21.73	21.73

Table A.77: Results of DKFMFAST effects on critical path: $d^{\Pi}_{\max}(H)$, on target T3 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	kHMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	0.15	0.15	0.46	0.46	1.61	1.61	0.08	0.01	8.68	8.27
mnist	4.17	4.17	2.92	2.92	6.53	6.53	2.78	2.45	11.11	11.11
mobilnet1	1.82	1.82	1.72	1.72	3.11	3.11	1.83	1.10	6.34	6.34
OneCore	0.32	0.32	0.11	0.11	1.96	1.96	0.17	0.01	6.60	6.60
PuLSAR	0.25	0.25	0.08	0.08	2.61	2.61	0.10	0.01	7.78	7.78
WasgaServer	0.01	0.01	0.47	0.47	2.56	2.56	0.08	0.01	7.82	7.82
bitonic_mesh	0.03	0.03	1.42	1.42	8.14	8.14	0.03	0.03	9.03	9.03
cholesky_bdti	0.97	0.97	0.97	0.97	3.25	3.25	0.90	0.21	6.18	6.18
dart	1.41	1.41	1.69	1.69	1.40	1.40	2.14	1.13	6.72	6.72
denoise	0.00	0.00	0.00	0.00	0.07	0.07	0.00	0.00	1.36	1.36
des90	1.42	1.42	1.42	1.42	7.20	7.20	0.18	0.03	8.22	8.22
xge_mac	2.50	2.50	3.98	3.98	5.72	5.72	4.28	3.98	7.02	7.02
cholesky_mc	1.08	1.08	1.13	1.13	3.09	3.09	0.92	0.26	7.98	7.98

Table A.78: Results of DKFMFAST effects on critical path: $d^{\Pi}_{\max}(H)$, on target T4 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	0.62	0.62	3.21	3.21	8.36	8.36	2.02	0.85	16.55	16.13
mnist	11.01	11.01	9.39	9.39	19.29	19.29	9.81	5.88	22.06	22.06
mobilnet1	5.35	5.35	4.95	4.95	12.57	12.57	4.19	1.83	13.05	13.05
OneCore	1.25	1.25	1.88	1.88	6.34	6.34	2.12	1.56	14.00	13.00
PuLSAR	2.92	2.92	2.20	2.20	8.68	8.68	2.23	1.07	28.98	28.96
WasgaServer	0.93	0.93	0.77	0.77	9.06	9.06	0.98	0.02	20.40	20.40
bitonic_mesh	2.10	2.10	2.94	2.94	24.21	23.49	2.21	2.10	16.60	16.60
cholesky_bdti	2.98	2.98	4.67	4.67	20.76	20.76	4.03	2.82	18.25	18.25
dart	3.93	3.93	4.21	4.21	5.89	5.89	4.71	3.37	11.50	11.48
denoise	0.00	0.00	0.02	0.02	1.80	1.80	0.00	0.00	5.55	5.55
des90	2.94	2.23	2.94	2.94	18.88	18.88	2.47	2.23	19.68	19.68
xge_mac	9.55	9.55	9.55	9.55	18.86	18.86	10.49	5.46	20.42	20.42
cholesky_mc	2.82	2.82	6.62	6.62	15.32	15.32	3.98	2.11	18.30	18.30

Table A.79: Results of DKFMFAST effects on critical path: $d^{\Pi}_{\max}(H)$, on target T5 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	kMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	0.99	0.99	3.83	3.83	8.57	8.57	2.33	0.85	29.35	29.35
mnist	12.63	12.63	12.54	12.54	28.81	28.81	14.24	9.30	33.76	33.76
mobilnet1	7.22	7.22	6.75	6.75	21.93	21.93	4.88	3.08	18.74	18.11
OneCore	2.18	2.18	2.24	2.24	5.39	5.39	2.76	1.54	10.37	9.46
PuLSAR	2.89	2.66	1.40	1.40	7.91	7.91	2.71	1.90	18.27	18.27
WasgaServer	0.84	0.84	1.06	1.06	14.21	14.21	1.18	0.43	24.87	24.87
bitonic_mesh	2.94	2.94	4.46	4.46	39.47	39.47	2.35	2.10	31.01	31.01
cholesky_bdti	4.67	4.67	4.67	4.67	19.78	19.78	4.22	2.82	16.46	16.46
dart	5.60	5.60	5.04	5.04	6.29	6.29	5.80	4.21	12.31	12.31
denoise	0.00	0.00	0.03	0.03	3.21	3.21	0.00	0.00	12.17	12.08
des90	3.67	3.67	4.46	4.46	19.24	19.24	2.34	2.23	18.92	18.92
xge_mac	13.28	13.20	13.20	13.20	21.64	21.64	14.45	8.42	29.64	29.64
cholesky_mc	4.78	4.78	6.62	6.62	17.17	17.17	4.52	2.82	16.46	16.46

Table A.80: Results of DKFMFAST effects on critical path: $d^{\Pi}_{\max}(H)$, on target T6 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	0.01	0.01	1.12	1.12	2.67	2.67	0.20	0.02	9.74	9.74
mnist	4.54	4.54	4.54	4.54	9.39	9.39	5.35	4.54	11.01	9.77
mobilnet1	2.23	2.23	1.83	1.83	6.34	6.34	1.92	0.58	5.72	5.72
OneCore	0.41	0.41	0.29	0.29	2.39	2.39	0.70	0.25	11.04	11.04
PuLSAR	0.44	0.44	0.21	0.21	2.58	2.58	0.56	0.48	9.63	9.63
WasgaServer	0.01	0.01	0.01	0.01	4.32	4.32	0.08	0.01	7.24	7.24
bitonic_mesh	1.42	1.42	1.42	1.42	12.83	12.83	0.09	0.03	9.79	9.79
cholesky_bdti	0.97	0.97	2.00	1.95	7.82	7.82	0.92	0.26	8.96	8.96
dart	2.53	2.53	2.25	2.25	2.28	2.28	2.78	2.25	7.28	7.28
denoise	0.00	0.00	0.01	0.01	1.65	1.65	0.00	0.00	1.53	1.53
des90	1.42	1.42	1.42	1.42	8.18	8.18	0.24	0.03	9.61	9.61
xge_mac	4.41	4.32	4.41	4.41	6.85	6.85	5.18	3.98	8.59	8.59
cholesky_mc	1.08	1.08	1.13	1.13	6.08	6.08	1.02	0.97	7.11	7.11

Connectivity cost

Results on connectivity cost presented in Chapter 6 are based on tables introduced in this subsection. Each table shows us effect of DKFMFAST on connectivity cost of partition produced by min-cut algorithms. These results have been used to make the figures presenting the relative connectivity cost of each partition. The following figures corresponds to the complementary results of the one presented in Chapter 6.

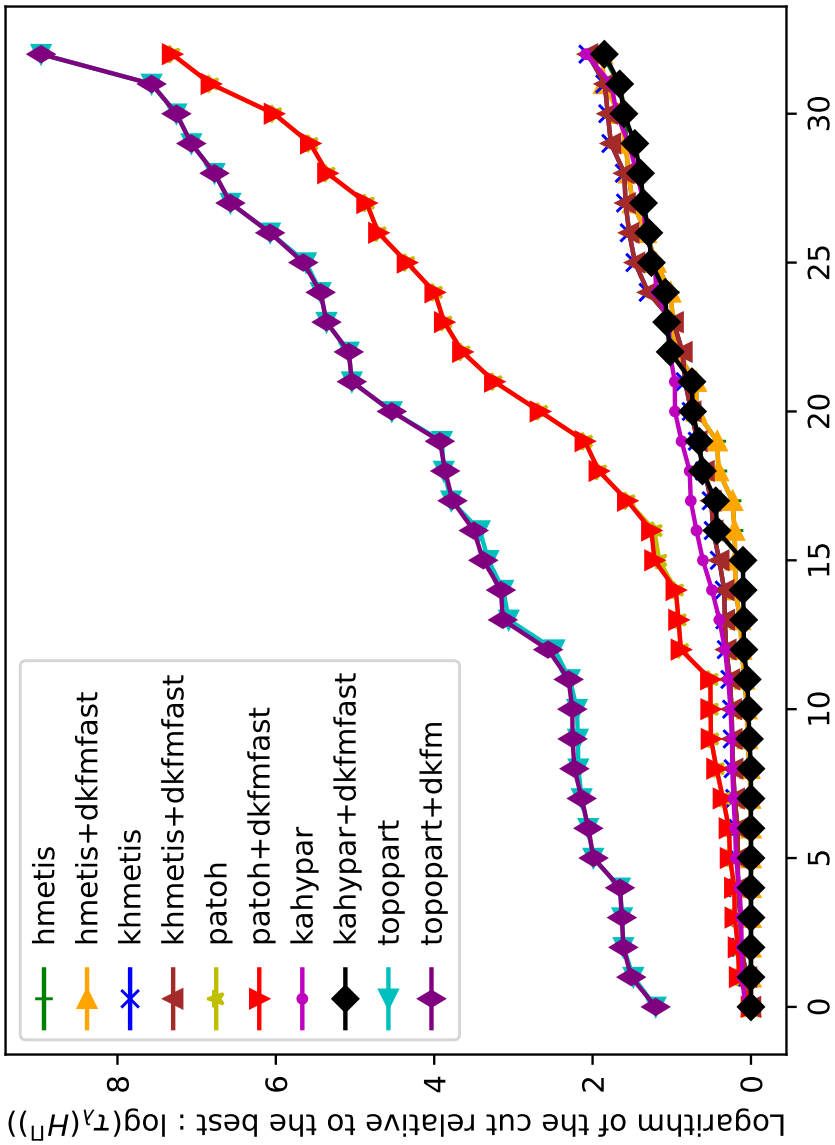


Figure A.9: Connectivity cost relative to the best in a logarithmic scale on target topology T1 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFMFAST.

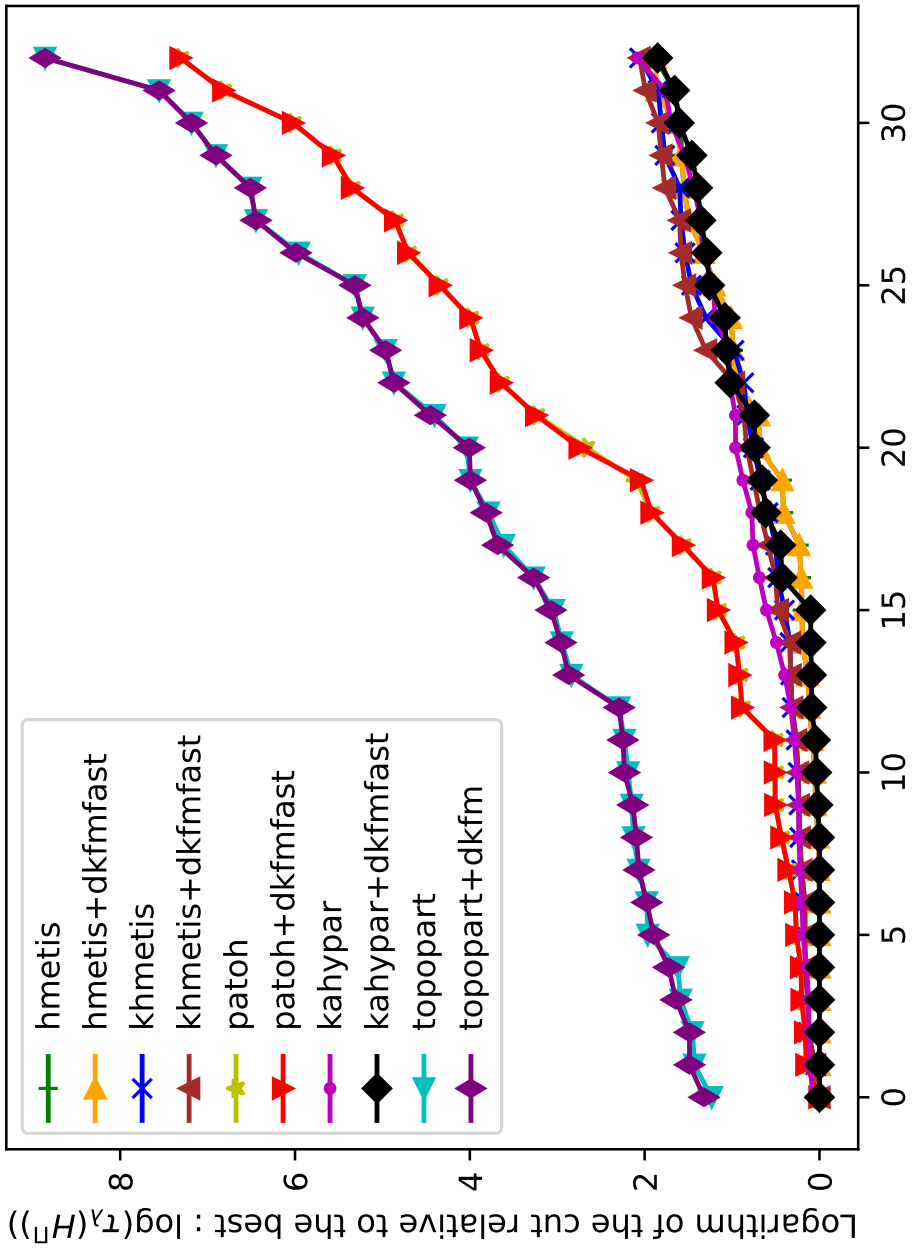


Figure A.10: Connectivity cost relative to the best in a logarithmic scale on target topology T2 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFMFAST.

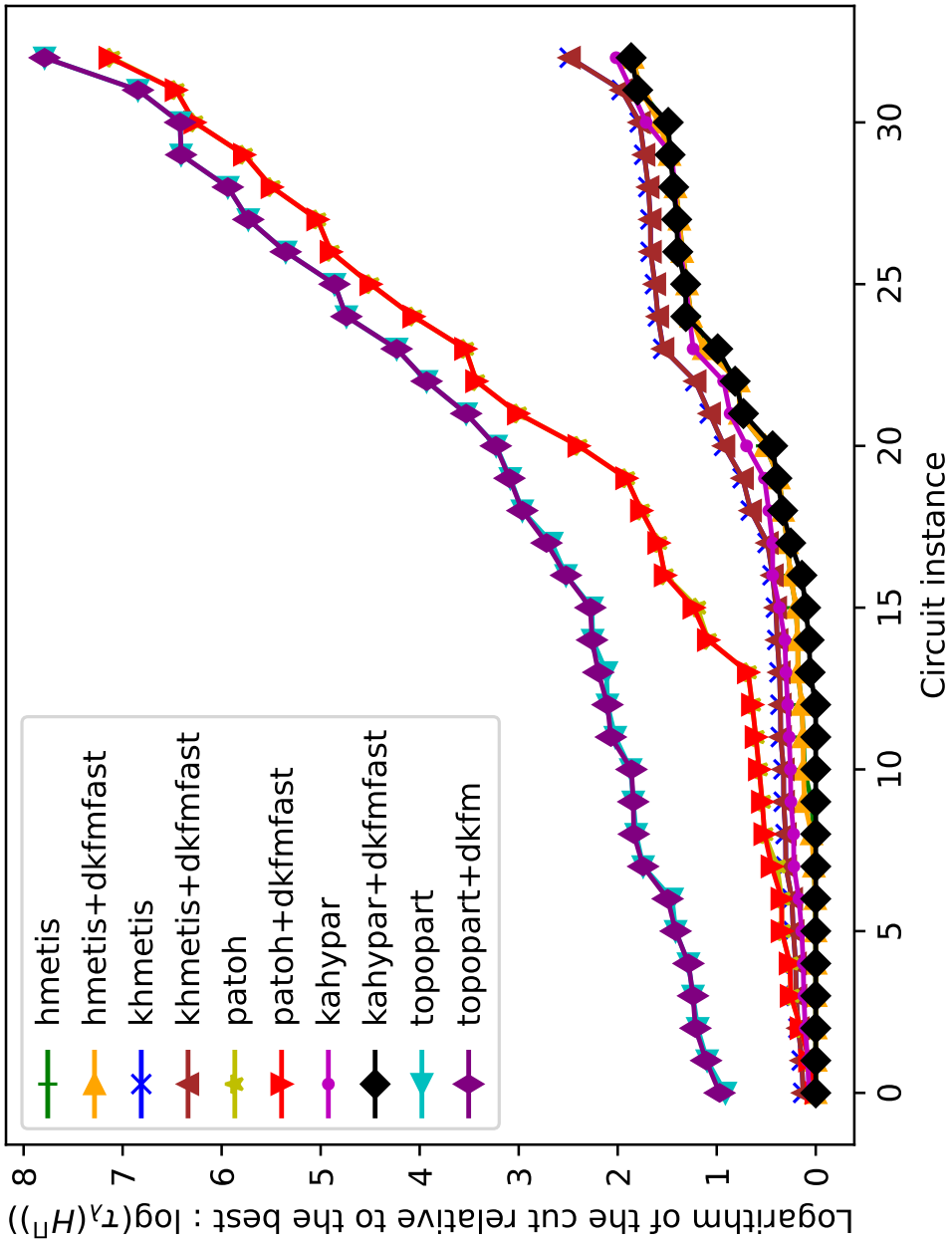


Figure A.11: Connectivity cost relative to the best in a logarithmic scale on target topology T4 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFMFAST.

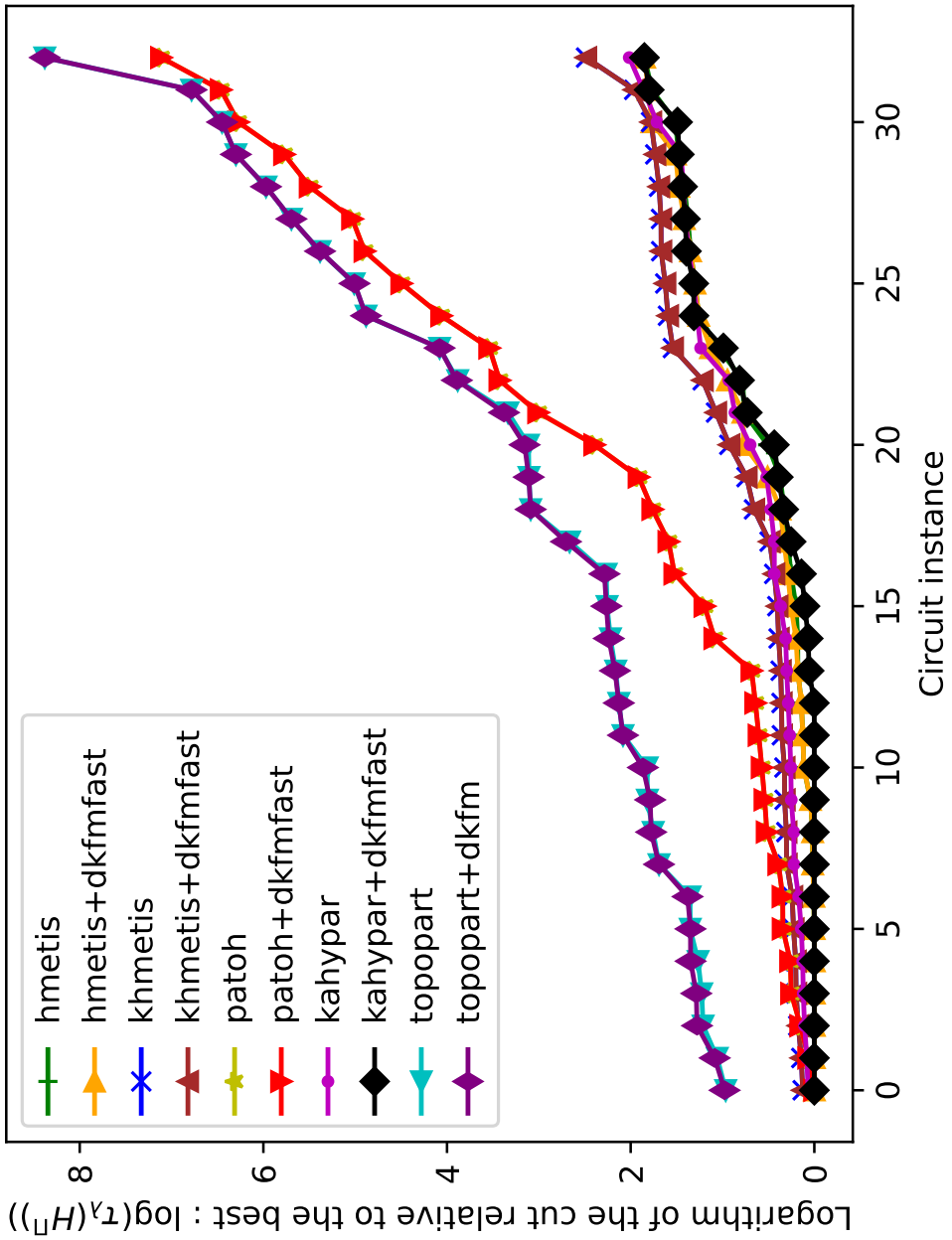


Figure A.12: Connectivity cost relative to the best in a logarithmic scale on target topology T5 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFMFAST.

Table A.81: Results of DKFMFAST effects on connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T1 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	15	15	19	19	20	20	14	15	30	30
b02	14	14	17	17	17	17	12	14	19	19
b03	18	18	29	29	22	24	18	19	58	58
b04	43	43	43	43	50	50	44	41	214	214
b05	33	33	37	37	39	39	41	34	328	328
b06	18	18	23	23	26	26	18	19	30	30
b07	40	40	50	50	52	52	45	40	133	133
b08	29	29	35	35	36	36	29	31	66	66
b09	24	24	23	23	29	29	24	26	79	79
b10	35	35	37	37	46	46	32	34	90	90
b11	45	45	58	58	60	60	52	47	228	228
b12	40	40	42	42	44	44	40	39	321	321
b13	2	2	5	5	6	6	6	7	76	76
b14	228	228	309	309	375	375	293	226	2709	2709
b17	153	153	187	187	236	236	227	155	7569	7569
b18	100	100	141	141	143	143	241	156	16153	16153
b19	375	375	286	286	145	145	380	306	33293	33293
b20	313	313	300	300	425	425	339	255	5434	5434
b21	296	296	297	297	391	391	284	235	5367	5367
b22	365	365	354	354	348	348	366	299	8258	8830

Table A.82: Results of DKFMFAST effects on connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T2 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
B01	15	15	19	19	20	20	14	15	30	30
B02	14	14	17	17	17	16	12	14	18	18
B03	18	18	29	29	22	22	18	19	64	64
B04	43	43	43	43	50	50	44	41	175	175
B05	33	33	37	37	39	39	41	34	241	241
B06	18	18	23	23	26	26	18	19	32	32
B07	40	40	50	50	52	52	45	40	137	146
B08	29	29	35	35	36	36	29	31	73	73
B09	24	24	23	23	29	29	24	26	78	78
B10	35	35	37	37	46	46	32	34	90	90
B11	45	45	58	58	60	60	52	47	190	190
B12	40	40	42	42	44	44	40	39	286	286
B13	2	2	5	5	6	6	6	7	65	65
B14	228	228	309	309	375	375	293	226	2243	2243
B17	153	153	187	187	236	236	227	155	5671	6149
B18	100	100	141	141	143	143	241	156	8231	8231
B19	375	375	286	286	145	145	380	306	20779	20779
B20	313	313	300	300	425	425	339	255	4861	4861
B21	296	296	297	297	391	391	284	235	4021	4422
B22	365	365	354	354	348	348	366	299	6264	6264

Table A.83: Results of DKFMFAST effects on connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T3 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	15	15	19	19	20	20	14	15	24	24
b02	14	14	17	17	17	17	12	14	18	18
b03	18	18	29	29	22	22	18	19	40	40
b04	43	43	43	43	50	50	44	41	120	120
b05	33	33	37	37	39	39	41	34	206	226
b06	18	18	23	23	26	26	18	19	25	25
b07	40	40	50	50	52	52	45	40	121	125
b08	29	29	35	35	36	36	29	31	55	55
b09	24	24	23	23	29	29	24	26	53	53
b10	35	35	37	37	46	46	32	34	76	76
b11	45	45	58	58	60	60	52	47	132	132
b12	40	40	42	42	44	48	40	39	194	206
b13	2	2	5	5	6	6	6	7	55	55
b14	228	228	309	309	375	375	293	226	2303	2303
b17	153	153	187	187	236	236	227	155	4440	4440
b18	100	100	141	141	143	143	241	156	6535	6535
b19	375	375	286	286	145	145	380	306	14537	15238
b20	313	313	300	300	425	425	339	255	3565	3985
b21	296	296	297	297	391	391	284	235	3598	3684
b22	365	365	354	354	348	348	366	299	5974	6139

Table A.84: Results of DKFMFAST effects on connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T4 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
B01	25	25	34	34	34	34	25	28	39	39
B02	24	24	27	27	25	25	21	23	31	31
B03	34	34	52	52	45	45	33	36	94	94
B04	74	74	91	91	104	104	79	74	244	244
B05	63	63	73	73	74	74	75	63	230	230
B06	30	30	39	39	41	41	29	30	44	44
B07	65	65	89	89	92	97	67	62	155	155
B08	51	51	64	64	61	61	47	47	106	106
B09	44	48	52	52	51	53	39	42	101	101
B10	62	62	72	72	85	85	57	59	118	118
B11	73	82	99	99	92	92	81	72	216	216
B12	81	81	86	86	93	93	81	80	307	317
B13	9	9	14	14	17	17	15	13	69	69
B14	437	437	463	463	637	637	513	380	2330	2330
B17	457	457	563	563	538	538	522	382	5497	5863
B18	273	273	390	390	292	292	395	290	5305	5305
B19	523	523	479	479	376	376	605	485	12863	14835
B20	630	630	653	653	811	811	607	472	3908	3908
B21	643	643	626	626	884	884	624	499	4752	5038
B22	779	779	803	803	983	983	693	537	6691	6691

Table A.85: Results of DKFMFAST effects on connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T5 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	25	25	34	34	34	34	25	28	37	37
b02	24	24	27	27	25	25	21	23	30	30
b03	34	34	52	52	45	45	33	36	85	91
b04	74	74	91	91	104	104	79	74	263	260
b05	63	63	73	73	74	74	75	63	214	214
b06	30	30	39	39	41	41	29	30	43	43
b07	65	65	89	89	92	92	67	62	164	164
b08	51	51	64	64	61	61	47	47	105	105
b09	44	48	52	52	51	51	39	42	96	96
b10	62	62	72	72	85	85	57	59	111	113
b11	73	73	99	99	92	92	81	72	207	207
b12	81	81	86	86	93	93	81	80	279	279
b13	9	9	14	14	17	17	15	13	77	77
b14	437	437	463	463	637	637	513	380	2198	2198
b17	457	457	563	563	538	538	522	382	5495	5495
b18	273	273	390	390	292	292	395	290	6015	6806
b19	523	523	479	479	376	376	605	485	8424	8424
b20	630	630	653	653	811	811	607	472	4512	4512
b21	643	643	626	626	884	884	624	499	4597	4597
b22	779	779	803	803	983	983	693	537	5181	5181

Table A.86: Results of DKFMFAST effects on connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T6 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
B01	25	25	34	34	34	34	25	28	45	45
B02	24	24	27	27	25	25	21	23	33	33
B03	34	34	52	52	45	45	33	36	73	73
B04	74	74	91	91	104	104	79	74	245	255
B05	63	63	73	73	74	74	75	63	250	275
B06	30	30	39	39	41	41	29	30	41	41
B07	65	65	89	89	92	92	67	62	142	142
B08	51	51	64	64	61	61	47	47	83	83
B09	44	44	52	52	51	51	39	42	86	86
B10	62	62	72	72	85	85	57	59	99	99
B11	73	76	99	99	92	92	81	72	225	225
B12	81	81	86	86	93	93	81	80	290	300
B13	9	9	14	14	17	17	15	13	87	87
B14	437	437	463	463	637	637	513	380	2363	2363
B17	457	457	563	563	538	538	522	382	5729	5867
B18	273	273	390	390	292	292	395	290	8302	8302
B19	523	523	479	479	376	376	605	485	16437	16437
B20	630	630	653	653	811	811	607	472	4740	4949
B21	643	643	626	626	884	884	624	499	4347	4347
B22	779	779	803	803	983	983	693	537	6241	6241

Table A.87: Results of DKFMFAST effects on connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T1 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	339	339	546	546	16334	16334	471	375	72204	72204
mnist	59	78	76	76	1558	1558	47	49	5266	5266
mobilnet1	124	124	134	134	21324	21324	174	125	71450	72379
OneCore	291	291	354	354	3662	3662	284	254	11224	11902
PuLSAR	550	550	673	673	24496	24496	738	451	69758	69758
WasgaServer	433	433	565	565	91152	91152	550	475	310204	310204
bitonic_mesh	211	211	231	231	71051	71051	180	184	92041	92041
cholesky_bdti	309	309	289	289	24165	24165	318	307	68608	68608
dart	231	231	220	220	8449	8449	279	241	20604	20604
denoise	8	8	9	9	611	611	63	7	62171	62389
des90	172	172	215	215	33674	33674	155	159	52052	52052
xge_mac	84	84	92	92	647	647	83	89	1226	1296
cholesky_mc	170	170	203	203	12474	12474	244	208	31098	31098

Table A.88: Results of DKFMFAST effects on connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T2 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	knMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	339	339	546	546	16334	16334	471	375	69043	69043
mnist	59	78	76	76	1558	1558	47	49	4701	4701
mobilnet1	124	124	134	134	21324	21324	174	125	54896	54896
OneCore	291	291	354	354	3662	3961	284	254	11244	11670
PuLSAR	550	550	673	3307	24496	24496	738	451	58753	59050
WasgaServer	433	433	565	565	91152	91152	550	475	273364	273364
bitonic_mesh	211	211	231	231	71051	71051	180	184	90752	90752
cholesky_bdti	309	309	289	289	24165	24165	318	307	58323	58323
dart	231	231	220	220	8449	8449	279	241	11980	11980
denoise	8	8	9	9	611	611	63	7	55994	57439
des90	172	172	215	215	33674	33674	155	158	48460	48460
xge_mac	84	84	92	92	647	647	83	89	1433	1433
cholesky_mc	170	170	203	203	12474	12474	244	208	20981	20981

Table A.89: Results of DKFMFAST effects on connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T3 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	339	339	546	546	16334	16334	471	375	40149	41016
mnist	59	59	76	76	1558	1558	47	49	3929	3929
mobilnet1	124	124	134	134	21324	21324	174	125	57561	57561
OneCore	291	291	354	354	3662	3662	284	254	8019	8019
PuLSAR	550	550	673	673	24496	24496	738	451	50737	50737
WasgaServer	433	433	565	565	91152	91152	550	475	121510	121510
bitonic_mesh	211	211	231	231	71051	71051	180	184	43656	43656
cholesky_bdti	309	309	289	289	24165	24165	318	307	34461	34461
dart	231	231	220	220	8449	8449	279	241	10362	10362
denoise	8	8	9	9	611	611	63	7	2576	2576
des90	172	172	215	215	33674	33674	155	158	23958	23958
xge_mac	84	84	92	92	647	647	83	89	733	733
cholesky_mc	170	170	203	203	12474	12474	244	208	16160	16160

Table A.90: Results of DKFMFAST effects on connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T4 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	knMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	553	553	1066	1066	32142	32142	724	636	63339	63339
mnist	144	144	203	203	3370	3370	153	157	4850	4850
mobilnet1	261	261	353	353	34894	34894	402	364	80351	80351
OneCore	697	697	884	884	6266	6266	712	571	12495	13083
PuLSAR	1506	1506	1813	1813	38721	38721	1670	1257	86575	86893
WasgaServer	752	752	907	907	163202	163202	773	671	254302	254302
bitonic_mesh	441	441	499	499	131346	133836	383	397	100498	100498
cholesky_bdti	667	667	672	672	60250	60250	731	733	69748	69748
dart	397	397	443	443	13440	13440	434	390	19873	20834
denoise	14	14	169	169	4536	4536	105	15	34050	34324
des90	379	407	423	423	53545	53545	357	370	50801	50801
xge_mac	222	222	297	297	1133	1133	220	228	1414	1414
cholesky_mc	291	291	394	394	20750	20750	321	305	28614	28614

Table A.91: Results of DKFMFAST effects on connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T5 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	553	553	1066	1066	32142	32142	724	636	72999	73624
mnist	144	144	203	203	3370	3370	153	157	5642	5642
mobilnet1	261	261	353	353	34894	34894	402	364	77503	78250
OneCore	697	697	884	884	6266	6266	712	571	12854	13330
PuLSAR	1506	3239	1813	1813	38721	38721	1670	1257	74412	74412
WasgaServer	752	752	907	907	163202	163202	773	671	263093	263093
bitonic_mesh	441	441	499	499	131346	131346	383	397	94376	94376
cholesky_bdti	667	667	672	672	60250	60250	731	723	71894	71894
dart	397	397	443	443	13440	13440	434	390	18974	18974
denoise	14	14	169	169	4536	4536	105	15	61577	61579
des90	379	379	423	423	53545	53545	357	370	45499	45499
xge_mac	222	342	297	297	1133	1133	220	228	1577	1577
cholesky_mc	291	291	394	394	20750	20750	321	305	29467	29467

Table A.92: Results of DKFMFAST effects on connectivity cost: $f_\lambda(H^\Pi) \times 10^3$, on target T6 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	knMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	553	553	1066	1066	32142	32142	724	636	48723	48723
mnist	144	144	203	203	3370	3370	153	157	4748	4728
mobilnet1	261	261	353	353	34894	34894	402	364	66785	66785
OneCore	697	697	884	884	6266	6266	712	571	13627	13627
PuLSAR	1506	1506	1813	1813	38721	38721	1670	1257	72032	72032
WasgaServer	752	752	907	907	163202	163202	773	671	147895	147895
bitonic_mesh	441	441	499	499	131346	131346	383	397	46519	46519
cholesky_bdti	667	667	672	1176	60250	60250	731	723	47576	47576
dart	397	397	443	443	13440	13440	434	390	11958	11958
denoise	14	14	169	169	4536	4536	105	15	3186	3186
des90	379	379	423	423	53545	53545	357	370	40545	40545
xge_mac	222	342	297	297	1133	1133	220	228	1125	1125
cholesky_mc	291	291	394	394	20750	20750	321	305	19019	19019

Balance cost

In Chapter 6, we presented a comparison on vertex weight balance of partition. For this purpose, we use the results presented in the following tables. Each table shows us effect of DKFMFAST on balance cost of partition produced by min-cut algorithms. The following figures corresponds to the complementary results of the one presented in Chapter 6.

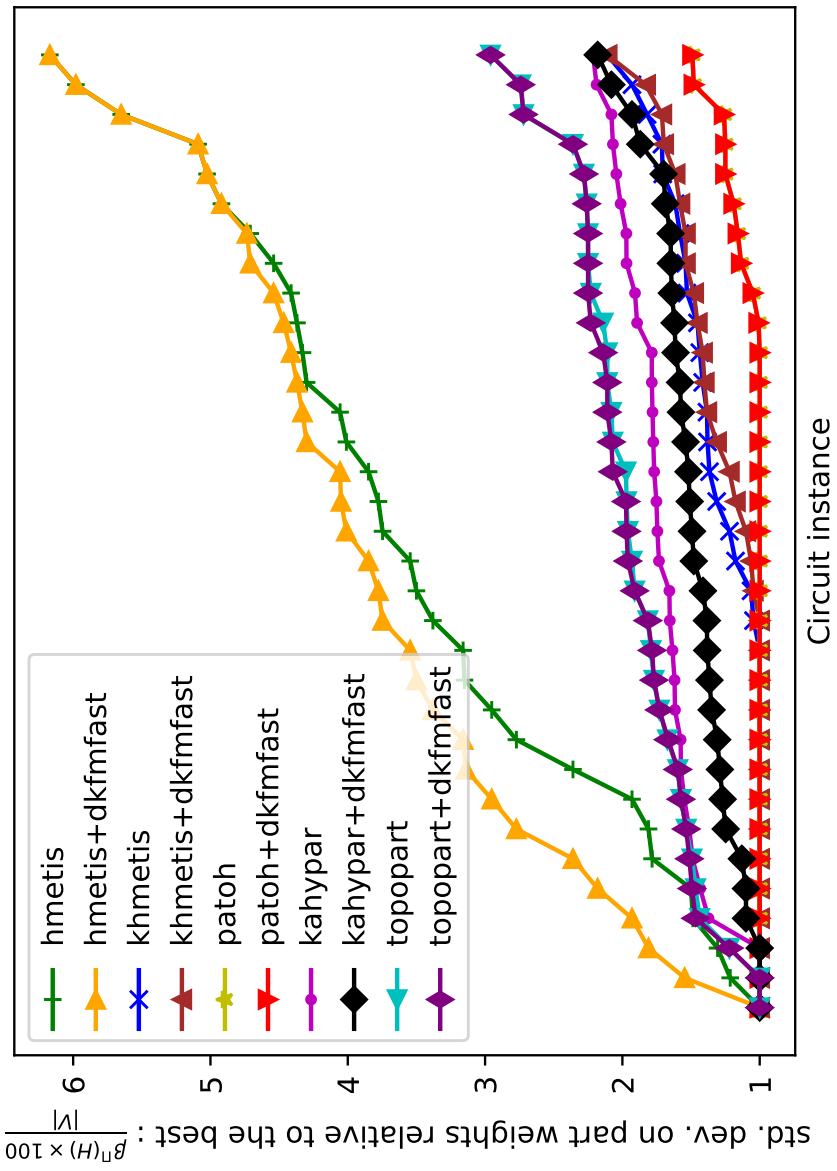


Figure A.13: Balance cost relative to the best on target topology T1 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFMFAST.

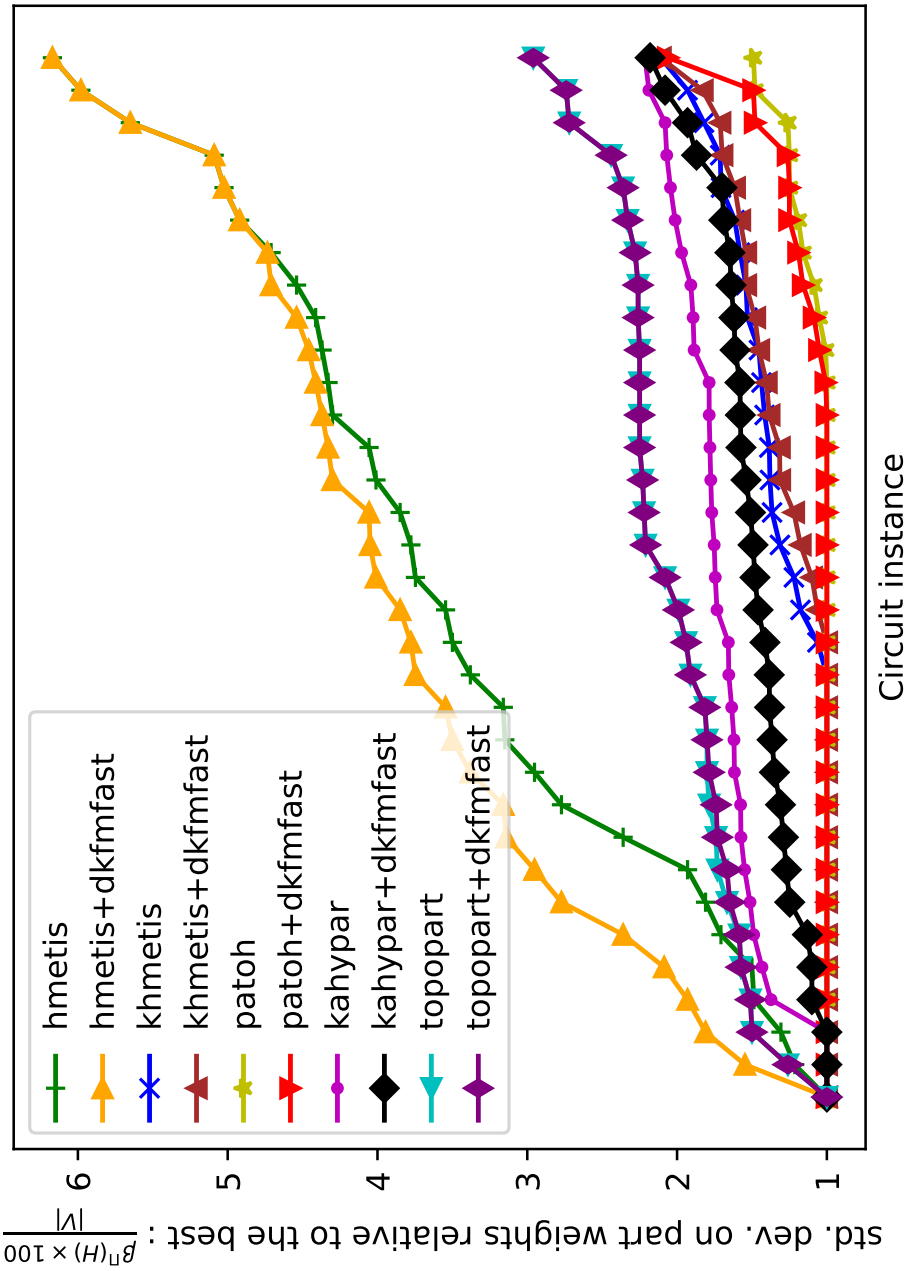


Figure A.14: Balance cost relative to the best on target topology T2 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFMFAST.

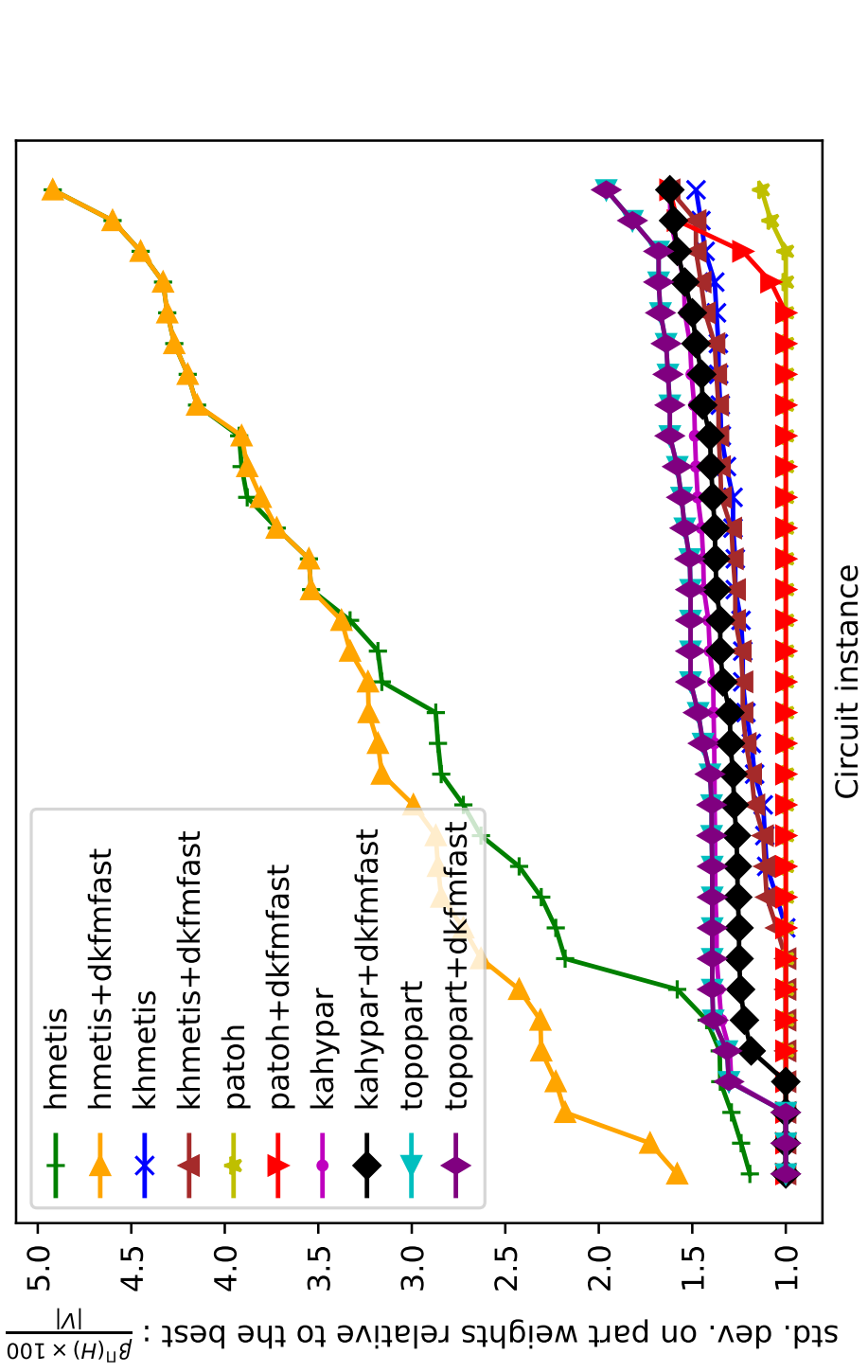


Figure A.15: Balance cost relative to the best on target topology T4 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFMFAST.

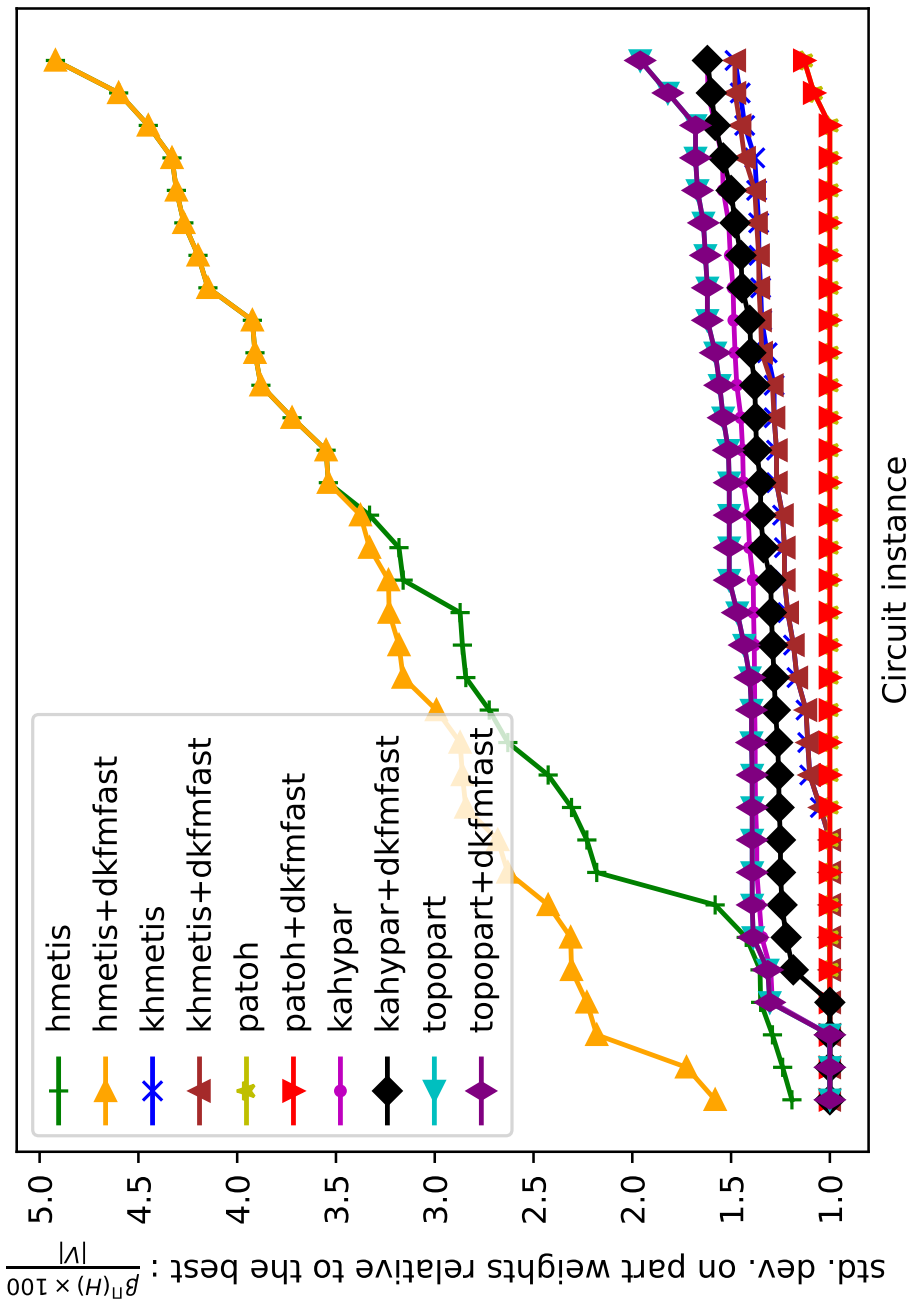


Figure A.16: Balance cost relative to the best on target topology T5 for hMetis, khMetis, PaToH, KaHyPar, TopoPart, and DKFMFAST.

Table A.93: Results of DKFMFAST effects on balance cost: $\beta(H^{\Pi})$, on target T1 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
B01	4.92	4.92	1.00	1.00	1.00	1.00	1.00	1.00	2.96	2.96
B02	4.33	4.33	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
B03	1.00	1.00	1.00	1.00	1.00	1.00	1.62	1.62	2.23	2.23
B04	1.93	1.93	1.53	1.00	1.00	1.00	2.01	1.13	2.07	2.07
B05	4.30	4.30	1.00	1.10	1.19	1.19	1.79	1.10	1.97	1.97
B06	6.17	6.17	1.00	1.00	1.00	1.00	1.00	1.00	2.72	2.72
B07	6.18	6.18	1.68	1.23	1.23	1.23	2.04	1.68	2.35	2.35
B08	3.16	3.16	1.54	1.54	1.00	1.00	2.08	2.08	2.08	2.08
B09	5.65	5.65	1.58	1.58	1.00	1.00	2.04	1.58	2.74	2.74
B10	4.37	4.37	1.00	1.48	1.48	1.48	1.58	1.48	1.96	1.96
B11	5.09	5.09	1.38	1.00	1.26	1.26	1.97	1.51	2.28	2.28
B12	4.71	4.71	1.93	1.83	1.00	1.00	2.07	1.93	2.11	2.11
B13	4.54	4.54	1.82	1.54	1.00	1.00	1.51	1.27	2.36	2.36
B14	5.56	5.56	2.15	2.14	1.26	1.26	2.08	1.65	2.23	2.23
B17	4.13	4.13	2.14	2.09	1.49	1.49	2.22	2.06	1.82	1.82
B18	1.96	2.32	1.97	1.97	1.50	1.50	2.15	2.12	2.20	2.20
B19	1.98	2.42	1.16	1.16	1.26	1.26	2.19	1.83	1.11	1.11
B20	2.13	5.75	1.67	1.67	1.42	1.42	2.20	2.12	2.18	2.18
B21	1.68	5.35	1.13	1.13	1.31	1.31	1.98	1.41	2.23	2.23
B22	1.64	6.03	2.18	2.18	1.35	1.35	2.18	2.05	1.94	2.25

Table A.94: Results of DKFMFAST effects on balance cost: $\beta(H^{\Pi})$, on target T2 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	4.92	4.92	1.00	1.00	1.00	1.00	1.00	1.00	2.96	2.96
b02	4.33	4.33	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
b03	1.00	1.00	1.00	1.00	1.00	1.00	1.62	1.62	2.23	2.23
b04	1.93	1.93	1.53	1.00	1.00	1.00	2.01	1.13	2.33	2.33
b05	4.30	4.30	1.00	1.10	1.19	1.19	1.79	1.10	2.26	2.26
b06	6.17	6.17	1.00	1.00	1.00	1.00	1.00	1.00	2.72	2.72
b07	6.18	6.18	1.68	1.23	1.23	1.23	2.04	1.68	2.35	2.35
b08	3.16	3.16	1.54	1.54	1.00	1.00	2.08	2.08	2.08	2.08
b09	5.65	5.65	1.58	1.58	1.00	1.00	2.04	1.58	2.74	2.74
b10	4.37	4.37	1.00	1.48	1.48	1.48	1.58	1.48	2.44	2.44
b11	5.09	5.09	1.38	1.00	1.26	1.26	1.97	1.51	2.28	2.28
b12	4.71	4.71	1.93	1.83	1.00	1.00	2.07	1.93	2.21	2.21
b13	4.54	4.54	1.82	1.54	1.00	1.00	1.51	1.27	2.36	2.36
b14	5.56	5.56	2.15	2.14	1.26	1.26	2.08	1.65	2.25	2.25
b17	4.13	4.13	2.14	2.09	1.49	1.49	2.22	2.06	2.25	2.25
b18	1.96	2.32	1.97	1.97	1.50	1.50	2.15	2.12	1.89	1.89
b19	1.98	2.42	1.16	1.16	1.26	1.26	2.19	1.83	2.25	2.25
b20	2.13	5.75	1.67	1.67	1.42	1.42	2.20	2.12	2.25	2.25
b21	1.68	5.35	1.13	1.13	1.31	1.31	1.98	1.41	2.25	2.25
b22	1.64	6.02	2.18	2.18	1.35	1.35	2.18	1.97	2.25	2.25

Table A.95: Results of DKFMFAST effects on balance cost: $\beta(H^{\Pi})$, on target T3 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
B01	4.92	4.92	1.00	1.00	1.00	1.00	1.00	1.00	2.96	2.96
B02	4.33	4.33	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
B03	1.00	1.00	1.00	1.00	1.00	1.00	1.62	1.62	2.23	2.23
B04	1.93	1.93	1.53	1.00	1.00	1.00	2.01	1.13	2.20	2.20
B05	4.30	4.30	1.00	1.10	1.19	1.19	1.79	1.10	2.26	2.16
B06	6.17	6.17	1.00	1.00	1.00	1.00	1.00	1.00	2.72	2.72
B07	6.18	6.18	1.68	1.23	1.23	1.23	2.04	1.68	2.35	2.35
B08	3.16	3.16	1.54	1.54	1.00	1.00	2.08	2.08	1.54	1.54
B09	5.65	5.65	1.58	1.58	1.00	1.00	2.04	1.58	2.74	2.74
B10	4.37	4.37	1.00	1.48	1.48	1.48	1.58	1.48	2.44	2.44
B11	5.09	5.09	1.38	1.00	1.26	1.26	1.97	1.51	2.28	2.28
B12	4.71	4.71	1.93	1.83	1.00	1.19	2.07	1.93	2.21	1.83
B13	4.54	4.54	1.82	1.54	1.00	1.00	1.51	1.27	2.09	2.09
B14	5.56	5.56	2.15	2.14	1.26	1.26	2.08	1.65	2.25	2.25
B17	4.13	4.13	2.14	2.09	1.49	1.49	2.22	2.06	2.25	2.25
B18	1.96	2.32	1.97	1.97	1.50	1.50	2.15	2.12	2.25	2.25
B19	1.98	2.42	1.16	1.16	1.26	1.26	2.19	1.83	2.13	2.25
B20	2.13	5.75	1.67	1.67	1.42	1.42	2.20	2.12	2.22	2.25
B21	1.68	5.35	1.13	1.13	1.31	1.31	1.98	1.41	2.25	2.25
B22	1.64	6.03	2.18	2.18	1.35	1.35	2.18	1.97	2.25	2.25

Table A.96: Results of DKFMFAST effects on balance cost: $\beta(H^{\Pi})$, on target T4 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	4.92	4.92	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
b02	4.33	4.33	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
b03	2.23	2.23	1.00	1.62	1.00	1.00	1.62	1.62	1.62	1.62
b04	2.86	2.86	1.27	1.27	1.00	1.00	1.50	1.40	1.67	1.67
b05	3.91	3.91	1.00	1.00	1.00	1.00	1.51	1.39	1.68	1.68
b06	4.45	4.45	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
b07	4.60	4.60	1.23	1.23	1.00	1.23	1.45	1.45	1.68	1.68
b08	3.16	3.16	1.00	1.00	1.00	1.00	1.38	1.00	1.54	1.54
b09	3.33	3.33	1.00	1.00	1.00	1.58	1.58	1.58	1.58	1.58
b10	3.88	3.88	1.48	1.48	1.00	1.00	1.48	1.48	1.96	1.96
b11	3.55	3.55	1.38	1.38	1.00	1.00	1.47	1.38	1.64	1.64
b12	3.13	3.13	1.56	1.56	1.09	1.09	1.54	1.37	1.65	1.65
b13	4.27	4.27	1.27	1.27	1.00	1.00	1.54	1.54	1.82	1.82
b14	4.08	3.96	1.37	1.54	1.04	1.04	1.61	1.56	1.62	1.62
b17	1.51	3.95	1.50	1.51	1.17	1.17	1.62	1.61	1.63	1.63
b18	1.49	2.89	1.48	1.48	1.25	1.25	1.61	1.58	1.63	1.63
b19	1.40	1.95	1.39	1.39	1.13	1.13	1.56	1.38	1.63	1.63
b20	1.58	3.78	1.58	1.58	1.17	1.17	1.59	1.52	1.63	1.63
b21	1.58	3.59	1.51	1.51	1.11	1.11	1.60	1.56	1.63	1.63
b22	1.57	3.47	1.59	1.59	1.16	1.16	1.59	1.45	1.63	1.63

Table A.97: Results of DKFMFAST effects on balance cost: $\beta(H^{\Pi})$, on target T5 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
B01	4.92	4.92	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
B02	4.33	4.33	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
B03	2.23	2.23	1.00	1.00	1.00	1.00	1.62	1.62	1.62	1.62
B04	2.86	2.86	1.27	1.27	1.00	1.00	1.50	1.40	1.67	1.67
B05	3.91	3.91	1.00	1.00	1.00	1.00	1.51	1.29	1.68	1.68
B06	4.45	4.45	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
B07	4.60	4.60	1.23	1.23	1.00	1.00	1.45	1.45	1.68	1.68
B08	3.16	3.16	1.00	1.00	1.00	1.00	1.38	1.00	1.54	1.54
B09	3.33	3.33	1.00	1.00	1.00	1.00	1.58	1.58	1.58	1.58
B10	3.88	3.88	1.48	1.48	1.00	1.00	1.48	1.48	1.96	1.96
B11	3.55	3.55	1.38	1.38	1.00	1.00	1.47	1.38	1.64	1.64
B12	3.13	3.13	1.56	1.56	1.09	1.09	1.54	1.37	1.65	1.65
B13	4.27	4.27	1.27	1.27	1.00	1.00	1.54	1.54	1.82	1.82
B14	4.08	4.08	1.37	1.54	1.04	1.04	1.61	1.56	1.62	1.62
B17	1.51	3.95	1.50	1.51	1.17	1.17	1.62	1.61	1.63	1.63
B18	1.49	2.89	1.48	1.48	1.25	1.25	1.61	1.58	1.63	1.63
B19	1.40	1.95	1.39	1.39	1.13	1.13	1.56	1.38	1.62	1.62
B20	1.58	3.78	1.58	1.58	1.17	1.17	1.59	1.52	1.63	1.63
B21	1.58	3.59	1.51	1.51	1.11	1.11	1.60	1.56	1.63	1.63
B22	1.57	3.47	1.59	1.59	1.16	1.16	1.59	1.45	1.63	1.63

Table A.98: Results of DKFMFAST effects on balance cost: $\beta(H^{\Pi})$, on target T6 for circuits in ITC set.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
b01	4.92	4.92	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
b02	4.33	4.33	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
b03	2.23	2.23	1.00	1.00	1.00	1.00	1.62	1.62	1.62	1.62
b04	2.86	2.86	1.27	1.27	1.00	1.00	1.50	1.40	1.67	1.67
b05	3.91	3.91	1.00	1.00	1.00	1.00	1.51	1.39	1.68	1.68
b06	4.45	4.45	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
b07	4.60	4.60	1.23	1.23	1.00	1.00	1.45	1.45	1.68	1.68
b08	3.16	3.16	1.00	1.00	1.00	1.00	1.38	1.00	1.54	1.54
b09	3.33	3.33	1.00	1.00	1.00	1.00	1.58	1.58	1.58	1.58
b10	3.88	3.88	1.48	1.48	1.00	1.00	1.48	1.48	1.96	1.96
b11	3.55	3.55	1.38	1.38	1.00	1.00	1.47	1.38	1.64	1.64
b12	3.13	3.13	1.56	1.56	1.09	1.09	1.54	1.37	1.65	1.65
b13	4.27	4.27	1.27	1.27	1.00	1.00	1.54	1.54	1.82	1.82
b14	4.08	4.08	1.37	1.37	1.04	1.04	1.61	1.56	1.61	1.61
b17	1.51	1.51	1.50	1.50	1.17	1.17	1.62	1.61	1.63	1.63
b18	1.49	1.49	1.48	1.48	1.25	1.25	1.61	1.58	1.62	1.62
b19	1.40	1.40	1.39	1.39	1.13	1.13	1.56	1.38	1.62	1.62
b20	1.58	1.58	1.58	1.58	1.17	1.17	1.59	1.52	1.63	1.63
b21	1.58	1.58	1.51	1.51	1.11	1.11	1.60	1.56	1.62	1.62
b22	1.57	1.57	1.59	1.59	1.16	1.16	1.59	1.45	1.63	1.63

Table A.99: Results of DKFMFAST effects on balance cost: $\beta(H^{\Pi})$, on target T1 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	knMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	5.07	5.07	2.14	2.14	1.25	1.25	2.23	2.11	2.09	1.89
mnist	3.15	3.14	1.00	1.00	1.49	1.49	1.77	1.70	2.26	2.26
mobilnet1	3.66	3.66	1.24	1.24	1.25	1.25	2.19	2.00	2.25	2.25
OneCore	5.98	5.98	2.11	2.11	1.00	1.00	2.19	2.18	1.93	2.25
PuLSAR	4.91	4.91	1.84	1.84	1.30	1.30	2.13	1.80	2.25	2.25
WasgaServer	5.62	5.62	2.18	2.18	1.50	1.50	2.06	1.65	2.24	2.24
bitonic_mesh	3.38	3.38	1.00	1.00	1.25	1.25	1.74	1.35	2.14	2.14
cholesky_bdti	4.01	4.01	1.22	1.22	1.00	1.00	1.91	1.54	2.11	2.11
dart	1.81	1.81	1.00	1.00	1.25	1.25	2.21	1.87	2.25	2.25
denoise	2.95	2.95	1.73	1.73	1.25	1.25	2.17	2.05	2.24	2.23
des90	3.50	3.50	1.00	1.00	1.00	1.00	1.89	1.29	2.25	2.25
xge_mac	5.07	5.07	1.52	1.52	1.43	1.43	2.25	2.25	2.25	2.25
cholesky_mc	4.58	4.58	1.19	1.19	1.25	1.25	2.12	1.96	1.90	1.90

Table A.100: Results of DKFMFAST effects on balance cost: $\beta(H^{\Pi})$, on target T2 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	5.07	5.07	2.14	2.14	1.25	1.25	2.23	2.11	2.25	2.25
mnist	3.15	3.14	1.00	1.00	1.49	1.49	1.77	1.70	2.26	2.26
mobilnet1	3.66	3.66	1.24	1.24	1.25	1.25	2.19	2.00	2.25	2.25
OneCore	5.98	5.98	2.11	2.11	1.00	2.06	2.19	2.18	2.25	2.25
PuLSAR	4.91	4.91	1.84	1.71	1.30	1.30	2.13	1.80	2.25	2.25
WasgaServer	5.62	5.62	2.18	2.18	1.50	1.50	2.06	1.65	2.25	2.25
bitonic_mesh	3.38	3.38	1.00	1.00	1.25	1.25	1.74	1.35	2.22	2.22
cholesky_bdti	4.01	4.01	1.22	1.22	1.00	1.00	1.91	1.54	2.25	2.25
dart	1.81	1.81	1.00	1.00	1.25	1.25	2.21	1.87	2.25	2.25
denoise	2.95	2.95	1.73	1.73	1.25	1.25	2.17	2.05	2.25	2.06
des90	3.50	3.50	1.00	1.00	1.00	1.00	1.89	1.29	2.25	2.25
xge_mac	5.07	5.07	1.52	1.52	1.43	1.43	2.25	2.25	2.25	2.25
cholesky_mc	4.58	4.58	1.19	1.19	1.25	1.25	2.12	1.96	2.07	2.07

Table A.101: Results of DKFMFAST effects on balance cost: $\beta(H^{\Pi})$, on target T3 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	kHMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	5.07	5.07	2.14	2.14	1.25	1.25	2.23	2.11	1.45	1.43
mnist	3.15	3.15	1.00	1.00	1.49	1.49	1.77	1.72	2.26	2.26
mobilnet1	3.66	3.66	1.24	1.24	1.25	1.25	2.19	2.00	2.25	2.25
OneCore	5.98	5.98	2.11	2.11	1.00	1.00	2.19	2.18	2.25	2.25
PuLSAR	4.91	4.91	1.84	1.84	1.30	1.30	2.13	1.80	2.25	2.25
WasgaServer	5.62	5.62	2.18	2.18	1.50	1.50	2.06	1.65	2.24	2.24
bitonic_mesh	3.38	3.38	1.00	1.00	1.25	1.25	1.74	1.35	2.25	2.25
cholesky_bdti	4.01	4.01	1.22	1.22	1.00	1.00	1.91	1.54	1.55	1.55
dart	1.81	1.81	1.00	1.00	1.25	1.25	2.21	1.87	2.25	2.25
denoise	2.95	2.95	1.73	1.73	1.25	1.25	2.17	2.05	1.55	1.55
des90	3.50	3.50	1.00	1.00	1.00	1.00	1.89	1.29	2.25	2.25
xge_mac	5.07	5.07	1.52	1.52	1.43	1.43	2.25	2.25	2.25	2.25
cholesky_mc	4.58	4.58	1.19	1.19	1.25	1.25	2.12	1.96	1.87	1.87

Table A.102: Results of DKFMFAST effects on balance cost: $\beta(H^{\Pi})$, on target T4 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	4.48	4.48	1.57	1.57	1.08	1.08	1.61	1.56	1.63	1.63
mnist	1.96	1.96	1.30	1.30	1.24	1.24	1.61	1.54	1.63	1.63
mobilnet1	3.69	3.69	1.29	1.29	1.16	1.16	1.59	1.48	1.62	1.62
OneCore	4.87	4.87	1.56	1.56	1.16	1.16	1.61	1.55	1.62	1.62
PuLSAR	5.04	5.04	1.45	1.45	1.17	1.17	1.62	1.58	1.63	1.63
WasgaServer	4.14	4.14	1.59	1.59	1.17	1.17	1.57	1.50	1.63	1.63
bitonic_mesh	2.18	2.18	1.00	1.00	1.13	1.62	1.49	1.25	1.62	1.62
cholesky_bdti	2.62	2.62	1.31	1.31	1.08	1.08	1.55	1.40	1.63	1.63
dart	2.63	2.63	1.00	1.00	1.08	1.08	1.62	1.60	1.63	1.63
denoise	2.70	2.70	1.50	1.50	1.17	1.17	1.61	1.58	1.63	1.63
des90	3.07	3.07	1.19	1.19	1.08	1.08	1.52	1.28	1.63	1.63
xge_mac	3.16	3.16	1.30	1.30	1.16	1.16	1.61	1.59	1.61	1.61
cholesky_mc	4.02	4.02	1.26	1.26	1.08	1.08	1.49	1.36	1.63	1.63

Table A.103: Results of DKFMFAST effects on balance cost: $\beta(H^{\Pi})$, on target T5 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	kHMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	4.48	4.48	1.57	1.57	1.08	1.08	1.61	1.56	1.63	1.63
mnist	1.96	1.96	1.30	1.30	1.24	1.24	1.61	1.54	1.63	1.63
mobilnet1	3.69	3.69	1.29	1.29	1.16	1.16	1.59	1.48	1.62	1.62
OneCore	4.87	4.87	1.56	1.56	1.16	1.16	1.61	1.55	1.62	1.62
PuLSAR	5.04	5.04	1.45	1.45	1.17	1.17	1.62	1.58	1.63	1.63
WasgaServer	4.14	4.14	1.59	1.59	1.17	1.17	1.57	1.50	1.63	1.63
bitonic_mesh	2.18	2.18	1.00	1.00	1.13	1.13	1.49	1.25	1.62	1.62
cholesky_bdti	2.62	2.62	1.31	1.31	1.08	1.08	1.55	1.40	1.63	1.63
dart	2.63	2.63	1.00	1.00	1.08	1.08	1.62	1.60	1.63	1.63
denoise	2.70	2.70	1.50	1.50	1.17	1.17	1.61	1.58	1.63	1.63
des90	3.07	3.07	1.19	1.19	1.08	1.08	1.52	1.28	1.63	1.63
xge_mac	3.16	3.11	1.30	1.30	1.16	1.16	1.61	1.59	1.61	1.61
cholesky_mc	4.02	4.02	1.26	1.26	1.08	1.08	1.49	1.36	1.63	1.63

Table A.104: Results of DKFMFAST effects on balance cost: $\beta(H^{\Pi})$, on target T6 for circuits in Chipyard and Titan sets.

Instance	hMetis	+DKFM	khMetis	+DKFM	PaToH	+DKFM	KaHyPar	+DKFM	TopoPart	+DKFM
EightCore	4.48	4.48	1.57	1.57	1.08	1.08	1.61	1.56	1.61	1.61
mnist	1.96	1.96	1.30	1.30	1.24	1.24	1.61	1.54	1.63	1.63
mobilnet1	3.69	3.69	1.29	1.29	1.16	1.16	1.59	1.48	1.62	1.62
OneCore	4.87	4.87	1.56	1.56	1.16	1.16	1.61	1.55	1.62	1.62
PuLSAR	5.04	5.04	1.45	1.45	1.17	1.17	1.62	1.58	1.63	1.63
WasgaServer	4.14	4.14	1.59	1.59	1.17	1.17	1.57	1.50	1.62	1.62
bitonic_mesh	2.18	2.18	1.00	1.00	1.13	1.13	1.49	1.25	1.62	1.62
cholesky_bdti	2.62	2.62	1.31	1.31	1.08	1.08	1.55	1.40	1.60	1.60
dart	2.63	2.63	1.00	1.00	1.08	1.08	1.62	1.60	1.62	1.62
denoise	2.70	2.70	1.50	1.50	1.17	1.17	1.61	1.58	1.63	1.63
des90	3.07	3.07	1.19	1.19	1.08	1.08	1.52	1.28	1.62	1.62
xge_mac	3.16	3.11	1.30	1.30	1.16	1.16	1.61	1.59	1.61	1.61
cholesky_mc	4.02	4.02	1.26	1.26	1.08	1.08	1.49	1.36	1.61	1.61

References

References

[1] KaHyPar. 5.5.2

[2] Cristinel Ababei and Kia Bazargan. Statistical timing driven partitioning for VLSI circuits. In *Proceedings 2002 of the Design, Automation and Test in Europe Conference and Exhibition*, page 1109. IEEE, 2002. 2.4.1

[3] Cristinel Ababei and Kia Bazargan. Timing minimization by statistical timing hMetis-based partitioning. *Proceedings 2003 of the 16th International Conference on VLSI Design.*, 2003. 2.4.1

[4] Cristinel Ababei, Navaratnasothie Selvakkumaran, Kia Bazargan, and George Karypis. Multi-objective circuit partitioning for cutsize and path-based delay minimization. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ICCAD '02, pages 181–185, New York, NY, USA, November 2002. Association for Computing Machinery. (document), 1.2.1, 2.4.1, 4.2, 4.2.2, 4.2, 5.1

[5] Tarun Agarwal, Amit Sharma, and Laxmikant V. Kalé. Topology-aware task mapping for reducing communication contention on large parallel machines. In *Proceedings of the 20th IEEE International Parallel Distributed Processing Symposium*, 2006. 2.1.3

[6] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Engineering a direct k-way hypergraph partitioning algorithm. In *Proceedings of the 19th Workshop on Algorithm Engineering and Experiments, (ALENEX 2017)*, pages 28–42, 2017. 2.3.3, 6.1.3

[7] Charles J. Alpert, Jen-Hsin Huang, and Andrew B. Kahng. Multilevel circuit partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(8):655–667, August 1998. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 1.1.2

[8] Charles J. Alpert and Andrew B. Kahng. Recent directions in netlist partitioning: a survey. *Integration*, 19(1):1–81, August 1995. 1.3.3, 1, 1.3.3, 1.3.3

[9] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21, 2020. 3.2

[10] Robin Andre, Sebastian Schlag, and Christian Schulz. Memetic multilevel hypergraph partitioning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, pages 347–354, 2018. 2.3.3

[11] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the 16th annual ACM symposium on Parallelism in algorithms and architectures*, pages 120–124, 2004. (document), 2.2.1, 4.3.1

[12] Shawki Areibi. An integrated genetic algorithm with dynamic hill climbing for VLSI circuit partitioning. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) 2000*, pages 97–102, 2000. 2.4.2

[13] Shawki Areibi and Anthony Vannelli. Circuit partitioning using a Tabu search approach. In *Proceedings 1993 IEEE International Symposium on Circuits and Systems*, pages 1643–1646 vol.3, May 1993. 2.4.2

[14] Shawki Areibi and Anthony Vannelli. Tabu search: A meta heuristic for netlist partitioning. *Proceedings of Very Large Scale Integration Design (VLSID)*, 11(3):259–283, 2000. 2.4.2

[15] Shawki Areibi and Zhen Yang. Effective Memetic Algorithms for VLSI Design = Genetic Algorithms + Local Search + Multi-Level Clustering. *Evolutionary Computation*, 12(3):327–353, September 2004. 2.4.2

[16] Cevdet Aykanat, Berkant Barla Cambazoglu, and Bora Uçar. Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *Journal of Parallel and Distributed Computing*, 68(5):609–625, 2008. 2.3.2, 6.1.3

[17] Raul Banos, Consolación Gil, Maria Dolores Gil Montoya, and Julio Ortega Lopera. A parallel evolutionary algorithm for circuit partitioning. In *Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 365–371, 2003. 2.4.2

[18] Ranieri Baraglia, Raffaele Perego, José Ignacio Hidalgo, Juan Lanchares, and Francisco Tirado. A parallel compact genetic algorithm for multi-FPGA partitioning. In *Proceedings of the 9th Euromicro Workshop on Parallel and Distributed Processing, PDP 2001, 7-9 February 2001, Mantova, Italy*, pages 113–120. IEEE Computer Society, 2001. 2.4.2

[19] Stephen T. Barnard and Horst D. Simon. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Journal of Concurrency: Practice and experience*, 6(2):101–117, 1994. 2.4.1

[20] Dirk Behrens, Klaus Harbich, and Erich Barke. Hierarchical partitioning. In *Proceedings of International Conference on Computer Aided Design*, pages 470–477. IEEE, 1996. 2.4.1

[21] Claude Berge. *Graphs and hypergraphs*. Elsevier Science Ltd., 1985. 1.1, 1.1.1, 1.1.2, 1.1.2

[22] Karl-Eduard Berger. *Mapping of large task network on manycore architecture*. Thesis, Université Paris Saclay (COmUE), December 2015. 2.1.3

[23] Vaughn Betz and Jonathan Rose. VPR: A new packing, placement and routing tool for FPGA research. In *International Workshop on Field Programmable Logic and Applications*, pages 213–222. Springer, 1997. 2.4.2, 3.2.2

[24] Shahid H. Bokhari. On the mapping problem. *Journal of IEEE Transactions on Computers*, 30(3):207–214, 1981. 2.1.3

[25] Pierre Bonami, Viet Hung Nguyen, Michel Klein, and Michel Minoux. On the solution of a graph partitioning problem under capacity constraints. In *Proceedings of the Combinatorial Optimization: Second International Symposium, ISCO 2012, Athens, Greece, April 19-21, 2012, Revised Selected Papers 2*, pages 285–296. Springer, 2012. 5.3, 5.3.2

[26] Daniel R. Brasen, Jean-Pierre Hiol, and Gabriele Saucier. Finding best cones from random clusters for FPGA package partitioning. In *Proceedings of ASP-DAC'95/CHDL'95/VLSI'95 with EDA Technofair*, pages 799–804. IEEE, 1995. 2.4.1, 5.2.3

[27] Daniel R. Brasen and Gabriele Saucier. Using cone structures for circuit partitioning into FPGA packages. *Journal of IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(7):592–600, 1998. 2.4.1, 5.2.3

[28] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-programmable gate arrays*, volume 180. Springer Science & Business Media, 1992. (document)

[29] Thang Nguyen Bui and Curt Jones. A heuristic for reducing fill-in in sparse matrix factorization. Technical report, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1993. 2.4.1

[30] Raghu Burra and Dinesh Bhatia. Timing driven multi-FPGA board partitioning. In *Proceedings of the 11th International Conference on VLSI Design*, pages 234–237. IEEE, 1998. 5.2.2

[31] Ismail Bustany, Grigor Gasparyan, Andrew B. Kahng, Ioannis Koutis, Bodhisatta Pramanik, and Zhiang Wang. An open-source constraints-driven general partitioning multi-tool for vlsi physical design. In *In proceeding 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023. 2.3.4, 2.4.1, 2.4.1

[32] Andrew E. Caldwell, Andrew B. Kahng, and Igor L. Markov. Improved algorithms for hypergraph bipartitioning. In *In proceedings 2000. Design Automation Conference. (IEEE Cat. No.00CH37106)*, pages 661–666, 2000. 2.4.1

[33] Ümit V. Çatalyürek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *Journal of IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, July 1999. Conference Name: IEEE Transactions on Parallel and Distributed Systems. 4.1.1, 6.1.3

[34] Ümit V. Çatalyürek and Cevdet Aykanat. PATOH: partitioning tool for hypergraphs. In *Encyclopedia of parallel computing*, pages 1479–1487. Springer, 2011. 2.3.2, 2.4.1, 6.1.3

[35] Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. More Recent Advances in (Hyper)Graph Partitioning. Technical Report arXiv:2205.13202, arXiv, June 2022. arXiv:2205.13202 [cs]. 1.2.1, 2.4.1, 4.1.1, 4.2, 4.2.2, 5.1, 6.1.3

[36] CEA-LIST. N2D2 [online]. Available : https://github.com/cea-list/n2d2. 3.2.3

[37] Howard R. Charney and Donald L. Plato. Efficient partitioning of components. In *Proceedings of the 5th annual Design Automation Workshop*, pages 16–1, 1968. 1.3.3

[38] Ming-Hung Chen, Yao-Wen Chang, and Jun-Jie Wang. Performance-Driven Simultaneous Partitioning and Routing for Multi-FPGA Systems. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1129–1134, December 2021. ISSN: 0738-100X. (document), 2.4.1, 3.1

[39] Yongseok Cheon and Martin Derek F. Wong. Design hierarchy guided multilevel circuit partitioning. In *Proceedings of the 2002 International Symposium on Physical Design*, pages 30–35, 2002. 2.4.1

[40] Cédric Chevalier and François Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6-8):318–331, July 2008. 2.3.5

[41] Thiagarajan Chockalingam and Sanjit Arunkumar. A randomized heuristics for the mapping problem: The genetic approach. *Parallel computing*, 18(10):1157–1165, 1992. 2.1.3

[42] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. Distributed optimization by ant colonies. 01 1991. 2.4.2

[43] Jason Cong, Honching Li, and Chang Wu. Simultaneous circuit partitioning/clustering with retiming for performance optimization. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, page 460–465, New York, NY, USA, 1999. Association for Computing Machinery. 2.4.1

[44] Jason Cong, Zheng Li, and Rajive Bagrodia. Acyclic multi-way partitioning of boolean networks. In *Proceedings of the 31st annual design automation conference*, pages 670–675, 1994. 2.4.1, 5.1

[45] Jason Cong and Sung Kyu Lim. Performance driven multiway partitioning. In *Proceedings of the 37th Design Automation Conference. (IEEE Cat. No.00CH37106)*, pages 441–446, January 2000. 2.4.1

[46] Jason Cong and M'Lissa Smith. A parallel bottom-up clustering algorithm with applications to circuit partitioning in VLSI design. In *Proceedings of the 30th International Design Automation Conference*, pages 755–760, 1993. 2.4.1

[47] Jason Cong and Chang Wu. Global clustering-based performance-driven circuit partitioning. In *Proceedings of the 2002 International Symposium on Physical design*, ISPD '02, pages 149–154, New York, NY, USA, April 2002. Association for Computing Machinery. 2.4.1, 4.1.3

[48] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Chapter 21: Data structures for Disjoint Sets*. MIT Press Cambridge, 2001. 5.2.3

[49] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. 2022. 4.3.3

[50] Fulvio Corno, Matteo Sonza Reorda, and Giovanni Squillero. RT-level ITC'99 benchmarks and first ATPG results. *Journal of IEEE Design & Test of computers*, 17(3):44–53, 2000. 3.2, 3.2.1, 5.5.1

[51] Panayiotis Danassis, Kostas Siozios, and Dimitrios Soudris. ANT3D: Simultaneous partitioning and placement for 3-D FPGAs based on ant colony optimization. *Journal of IEEE Embedded Systems Letters*, 8(2):41–44, 2016. 2.4.2

[52] George B. Dantzig. *Linear programming and extensions*. Princeton university press, 1963. 2.4.1

[53] George B. Dantzig. Linear programming. *Operations research*, 50(1):42–47, 2002. 2.4.1

[54] Mehmet Deveci, Karen D. Devine, Kevin Pedretti, Mark A. Taylor, Sivasankaran Rajamanickam, and Ümit V. Çatalyürek. Geometric mapping of tasks to processors on parallel computers with mesh or torus networks. *Journal IEEE Transactions on Parallel and Distributed Systems*, 30(9):2018–2032, 2019. 2.1.3

[55] Mehmet Deveci, Kamer Kaya, Bora Uçar, and Ümit V. Çatalyürek. Hypergraph partitioning for multiple communication cost metrics: Model and methods. *Journal of Parallel and Distributed Computing*, 77:69–83, 2015. 1.3.1

[56] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Ümit V. Çatalyürek. Parallel hypergraph partitioning for scientific computing. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10–pp. IEEE, 2006. 6.1.3

[57] Ajit A. Diwan, Sanjeeva Rane, Sridhar Seshadri, and Sundararajarao Sudarshan. Clustering techniques for minimizing external path length. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 342–353, 1996. 2.2.2, 4.1.3, 4.3.2

[58] Wilm E. Donath, Reini J. Norman, Bhuwan K. Agrawal, Stephen E. Bello, Sang Yong Han, Jerome M. Kurtzberg, Paul Lowy, and Roger I. McMillan. Timing driven placement using complete path delays. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 84–89, june 1990. ISSN: 0738-100X. 2.4.1

[59] Zola Nailah Donovan. *Algorithmic Issues in some Disjoint Clustering Problems in Combinatorial Circuits*. Thesis dissertation, West Virginia University Libraries, January 2018. 2.4.1, 4.1.3, 4.2.1, 4.3.3, 4.3.3, 4.4.1, 4.4.2

[60] Zola Nailah Donovan, Vahan Mkrtchyan, and Kirubakran Subramani. Complexity issues in some clustering problems in combinatorial circuits. *arXiv:1412.4051 [cs]*, January 2017. arXiv: 1412.4051 version: 2. 2.2.2, 2.4.1, 4.1.3

[61] Zola Nailah Donovan, Kirubakran Subramani, and Vahan Mkrtchyan. Disjoint Clustering in Combinatorial Circuits. In Charles J. Colbourn, Roberto Grossi, and Nadia Pisanti, editors, *Combinatorial Algorithms*, Lecture Notes in Computer Science, pages 201–213, Cham, 2019. Springer International Publishing. 2.2.2, 2.4.1, 4.1.3, 4.3.1, 4.4.2

[62] Zola Nailah Donovan, Kirubakran Subramani, and Vahan Mkrtchyan. Analyzing Clustering and Partitioning Problems in Selected VLSI Models. *Theory of Computing Systems*, 64(7):1242–1272, October 2020. 2.4.1, 4.1.3, 4.4

[63] Marco Dorigo. The ant system: An autocatalytic optimizing process. In *Proceedings of the 1st European Conference on Artificial Life, Paris, France, 1991*, 1991. 2.4.2

[64] Marco Dorigo and Luca Maria Gambardella. Ant colonies for the travelling salesman problem. *Journal of Biosystems*, 43(2):73–81, 1997. 2.4.2

[65] Shantanu Dutt. New faster Kernighan-Lin-type graph-partitioning algorithms. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 370–377. IEEE, 1993. 6.1.1

[66] Shantanu Dutt and Wenyong Deng. A probability-based approach to VLSI circuit partitioning. In *Proceedings of the 33rd annual Design Automation Conference*, pages 100–105, 1996. 2.3.1, 2.4.1

[67] Shantanu Dutt and Wenyong Deng. VLSI circuit partitioning by cluster-removal using iterative improvement techniques. In *Proceedings of International Conference on Computer Aided Design*, pages 194–200. IEEE, 1996. 2.3.1, 2.4.1

[68] John M. Emmert and Dinesh K. Bhatia. Two-dimensional placement using tabu search. *Proceedings of the Very Large Scale Integration Design (VLSID)*, 12(1):13–23, 2001. 2.4.2

[69] John M. Emmert, Sandeep Lodha, and Dinesh K. Bhatia. On using tabu search for design automation of VLSI systems. *Journal of Heuristics*, 9:75–90, 2003. 2.4.2

[70] Leonhard Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, pages 128–140, 1741. 1.1

[71] Shimon Even. *Graph algorithms*. Cambridge University Press, 2011. 5.2.2

[72] Wen-Jong Fang and Allen C.-H. Wu. Multiway FPGA partitioning by fully exploiting design hierarchy. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(1):34–50, 2000. 2.4.1

[73] Umer Farooq and Bander A. Alzahrani. Exploring and optimizing partitioning of large designs for multi-FPGA based prototyping platforms. *Computing*, 102(11):2361–2383, 2020. 2.4.1, 2.4.1

[74] Umer Farooq, Imran Baig, Muhammad Khurram Bhatti, Habib Mehrez, Arun Kumar, and Manoj Gupta. Prototyping using multi-FPGA platform: A novel and complete flow. *Microprocessors and Microsystems*, 96:104751, 2023. 2.4.1

[75] Charles M. Fiduccia and Robert M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, pages 175–181, 1982. (document), 2.4.1, 2.4.1, 2.4.2, 6, 6.1.2, 6.1.2, 6.1.4

[76] Per-Olof Fjällström. *Algorithms for graph partitioning: A survey*. Linköping University Electronic Press, 1998. 5.3

[77] Lester Randolph Ford and Delbert R. Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8:399–404, 1956. 2.2.1

[78] Michael R. Garey and David S. Johnson. Computers and intractability. *A guide to the theory of NP-Completeness*, 1979. 2.1.3, 2.2.1, 3.1.1, 5.1

[79] Sandeep Singh Gill, Bhupesh Aneja, Rajeevan Chandel, and Ashwani Kumar Chandel. Simulated annealing based VLSI circuit partitioning for delay minimization. In *Proceedings of the 4th WSEAS international conference on Computational intelligence*, pages 60–63, 2010. 2.4.2

[80] Sandeep Singh Gill, Rajeevan Chandel, and Ashwani Chandel. Genetic algorithm based approach to circuit partitioning. *International Journal of Computer and Electrical Engineering*, 2(2):196, 2010. 2.4.2

[81] Fred Glover and Manuel Laguna. *Tabu search*. Springer, 1998. 2.4.2

[82] Mark Goldberg and Zevi Miller. A parallel algorithm for bisection width in trees. *Computers & Mathematics with Applications*, 15(4):259–266, 1988. 4.3.2

[83] Olivier Goldschmidt and Dorit S. Hochbaum. A Polynomial Algorithm for the k-cut Problem for Fixed k. *Mathematics of Operations Research*, 19(1):24–37, February 1994. Publisher: INFORMS. 2.2.1

[84] Ralph E. Gomory. Solving linear programming problems in integers. *Combinatorial Analysis*, 10:211–215, 1960. 2.4.1

[85] Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. Advanced flow-based multilevel hypergraph partitioning. In *Proceedings of the 18th International Symposium on Experimental Algorithms (SEA)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 11:1–11:15, 2020. 2.3.3

[86] Anael Grandjean, Johannes Langguth, and Bora Uçar. On optimal and balanced sparse matrix partitioning problems. In *Proceedings of 2012 IEEE International Conference on Cluster Computing*, pages 257–265. IEEE, 2012. 5.2.1

[87] Scott Hauck and Gaetano Borriello. An evaluation of bipartitioning techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(8):849–866, 1997. 2.4.1

[88] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, page 28–es, New York, NY, USA, 1995. Association for Computing Machinery. 2.4.1, 4.1.1

[89] Alexandra Henzinger, Alexander Noe, and Christian Schulz. ILP-based local search for graph partitioning. *ACM, Journal of Experimental Algorithmics (JEA)*, 25:1–26, jul 2020. 2.4.1, 5.3

[90] Monika Henzinger, Satish Rao, and Di Wang. Local flow partitioning for faster edge connectivity. *SIAM Journal on Computing*, 49(1):1–36, 2020. 2.2.1

[91] Julien Herrmann, Jonathan Kho, Bora Uçar, Kamer Kaya, and Ümit V. Çatalyürek. Acyclic partitioning of large directed acyclic graphs. In *2017 17th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID)*, pages 371–380. IEEE, 2017. 2.4.1, 5.1

[92] Julien Herrmann, M. Yusuf Ozkaya, Bora Uçar, Kamer Kaya, and Ümit V. Çatalyürek. Multilevel algorithms for acyclic partitioning of directed acyclic graphs. *SIAM Journal on Scientific Computing*, 41(4):A2117–A2145, 2019. 2.4.1, 5.1

[93] Tobias Heuer, Peter Sanders, and Sebastian Schlag. Network flow-based refinement for multilevel hypergraph partitioning. In *17th International Symposium on Experimental Algorithms (SEA 2018)*, pages 1:1–1:19, 2018. 2.3.3

[94] Tobias Heuer, Peter Sanders, and Sebastian Schlag. Network flow-based refinement for multilevel hypergraph partitioning. *Journal of Experimental Algorithmics (JEA)*, 24:1–36, 2019. 2.3.3, 6

[95] Tobias Heuer and Sebastian Schlag. Improving coarsening schemes for hypergraph partitioning by exploiting community structure. In *Proceedings of the 16th International Symposium on Experimental Algorithms, (SEA 2017)*, pages 21:1–21:19, 2017. 2.3.3, 4.1.1

[96] José Ignacio Hidalgo, Juan Lanchares, and Román Hermida. Partitioning and placement for multi-FPGA systems using genetic algorithms. In *Proceedings of the 26th Euromicro Conference. EUROMICRO 2000. Informatics: Inventing the Future*, volume 1, pages 204–211 vol.1, 2000. 2.4.2

[97] Torsten Hoefler and Marc Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proceedings of the International Conference on Supercomputing*, pages 75–84, 2011. 2.1.3

[98] Karla Hoffman and Manfred Padberg. Solving airline crew scheduling problems by branch-and-cut. *Management Science*, 39:657–682, 06 1993. 2.4.1

[99] John H. Holland. Genetic Algorithms and Adaptation. In Oliver G. Selfridge, Edwina L. Rissland, and Michael A. Arbib, editors, *Adaptive Control of Ill-Defined Systems*, pages 317–333. Springer US, Boston, MA, 1984. 2.4.1, 2.4.2

[100] John H. Holland. Genetic algorithms. *Scientific American*, 267(1):66–73, 1992. 2.4.1, 2.4.2

[101] Dennis J.-H. Huang and Andrew B. Kahng. Multi-way system partitioning into a single type or multiple types of FPGAs. In *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 140–145, 1995. 5.2.2

[102] Edmund Ihler, Dorothea Wagner, and Frank Wagner. Modeling hypergraphs by graphs with the same mincut properties. *Information Processing Letters*, 45(4):171–175, 1993. 1.3.3, 2.1.2

[103] Arthur. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, nov 1962. 3.1.2

[104] Andrew B. Kahng and Xu Xu. Local unidirectional bias for smooth cutsize-delay tradeoff in performance-driven bipartitioning. In *Proceedings of the 2003 international symposium on Physical design*, ISPD '03, pages 81–86, New York, NY, USA, April 2003. Association for Computing Machinery. 2.4.1

[105] Richard M. Karp. *Reducibility among combinatorial problems*. Springer, 1972. 2.4.1

[106] George Karypis. Metis: Unstructured graph partitioning and sparse matrix ordering system. *Technical report*, 1997. 2.3.1, 2.4.1

[107] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, March 1999. Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 2.4.1, 2.4.2, 6.1.3

[108] George Karypis and Vipin Kumar. Analysis of multilevel graph partitioning. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, pages 29–es, 1995. 4.4.2

[109] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998. 2.3.1, 2.4.1, 4.1.1

[110] George Karypis and Vipin Kumar. Hmetis: a hypergraph partitioning package. *ACM Transactions on Architecture and Code Optimization*, 1998. 4.4.2, 6.1.3

[111] George Karypis and Vipin Kumar. A hypergraph partitioning package. *Army HPC Research Center, Department of Computer Science & Engineering, University of Minnesota*, 1998. 2.3.1

[112] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. In *Proceedings of the 36th annual ACM/IEEE design automation conference*, pages 343–348, 1999. 2.3.1, 2.4.1, 6.1.3

[113] Brian W. Kernighan. Optimal sequential partitions of graphs. *Journal of the ACM (JACM)*, 18(1):34–40, 1971. 5.2.1

[114] Brian W. Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(2):291–307, 1970. 2.4.1, 6, 6.1.1, 6.1.1

[115] Scott Kirkpatrick, C. Daniel Gelatt Jr, and Mario P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983. 2.4.2

[116] Shad Kirmani, Jeonghyung Park, and Padma Raghavan. An embedded sectioning scheme for multiprocessor topology-aware mapping of irregular applications. *The International Journal of High Performance Computing Applications*, 31(1):91–103, 2017. 2.1.3

[117] Venkataramana Kommu and Irith Pomeranz. GAFPGA: genetic algorithm for FPGA technology mapping. In *Proceedings of the European Design Automation Conference 1993, EURO-DAC '93 with EURO-VHDL'93, Hamburg, Germany, September 20-24, 1993*, pages 300–305. IEEE Computer Society, 1993. 2.4.2

[118] Helena Krupnova, Ali Abbara, and Gabrièle Saucier. A Hierarchy-Driven FPGA Partitioning Method. page 4. 2.4.1

[119] Dorothy Kucar, Shawki Areibi, and Anthony Vannelli. Hypergraph partitioning techniques. *Dynamics of Continuous Discrete and Impulsive Systems Series A*, 11:339–368, 2004. 5.3

[120] Ailsa H. Land and Alison G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960. 2.4.1

[121] Eugene L. Lawler. Cutsets and partitions of hypergraphs. *Networks*, 3(3):275–285, 1973. 2.2.1

[122] Eugene L. Lawler, Karl N. Levitt, and James Turner. Module clustering to minimize delay in digital networks. *IEEE Transactions on Computers*, 100(1):47–57, 1969. 2.4.1, 4.1.2

[123] Chin Yang Lee. An algorithm for path connections and its applications. *IRE transactions on electronic computers*, (3):346–365, 1961. 5.2.1

[124] Ming Leng and Songnian Yu. An effective multi-level algorithm based on ANT colony optimization for bisecting graph. In *Proceedings ot Advances in Knowledge Discovery and Data Mining: 11th Pacific-Asia Conference, PAKDD 2007, Nanjing, China, May 22-25, 2007. Proceedings 11*, pages 138–149. Springer, 2007. 2.4.2

[125] Thomas Lengauer. Combinatorial algorithms for integrated circuit layout. JohnWiley & Sons. *Inc., New York*, 199, 1990. (document), 1.3.3, 2.1.2, 2.2.1, 5.1

[126] Jianmin Li, John Lillis, and Chung-Kuan Cheng. Linear decomposition algorithm for VLSI design applications. In Richard L. Rudell, editor, *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1995, San Jose, California, USA, November 5-9, 1995*, pages 223–228. IEEE Computer Society / ACM, 1995. 2.2.1

[127] Sin-Hong Liou, Sean Liu, Richard Sun, and Hung-Ming Chen. Timing Driven Partition for Multi-FPGA Systems with TDM Awareness. In *Proceedings of the 2020 International Symposium on Physical Design*, ISPD '20, pages 111–118, New York, NY, USA, March 2020. Association for Computing Machinery. (document), 2.4.1, 3.1, 3.2.2, 5.1, 5.4

[128] Sandeep Lodha and Dinesh Bhatia. Bipartitioning circuits using tabu search. In *Proceedings of the 11th Annual IEEE International ASIC Conference (Cat. No. 98TH8372)*, pages 223–227. IEEE, 1998. 2.4.2

[129] Robert Malcolm Macgregor. *On partitioning a graph: a theoretical and empirical study.* University of California, Berkeley, 1978. 4.3.2

[130] Pongstorn Maidee, Cristinel Ababei, and Kia Bazargan. Fast timing-driven partitioning-based placement for island style FPGAs. In *Proceedings of the 40th annual design automation conference*, pages 598–603, 2003. 2.4.2

[131] Ramachandran Manikandan, Ramalingam Parameshwaran, Prassanna Jayachandran, and K. R. Raju Sekar. A Study on Specific Computational Algorithms for VLSI Cell Partitioning Problems. pages 67–70, January 2019. 2.4.2

[132] Theodore Manikas and James T. Cain. Genetic Algorithms vs. Simulated Annealing: A Comparison of Approaches for Solving the Circuit Partitioning Problem. Technical report, 1996. 2.4.2

[133] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Partitioning complex networks via size-constrained clustering. In *Proceedings of the International Symposium on Experimental Algorithms*, pages 351–363. Springer, 2014. 5.1

[134] Jun'ichiro Minami, Tetsushi Koide, and Shin'ichi Wakabayashi. A circuit partitioning algorithm under path delay constraints. In *Proceedings of 1998 IEEE Asia-Pacific Conference on Circuits and Systems Microelectronics and Integrating System (APCCAS)*, pages 113–116. IEEE, 1998. 2.4.1

[135] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959. 5.2.1

[136] Orlando Moreira, Merten Popp, and Christian Schulz. Evolutionary multi-level acyclic graph partitioning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 332–339, 2018. 5.1

[137] Kevin E. Murray, Scott Whitty, Suya Liu, Jason Luu, and Vaughn Betz. TITAN : Enabling large and complex benchmarks in academic CAD. In *Proceedings of the 23rd International Conference on Field programmable Logic and Applications*, pages 1–8, Porto, Portugal, September 2013. IEEE. 3.2, 3.2.2

[138] Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992. 2.2.1

[139] Dang Phuong Nguyen, Michel Minoux, Viet Hung Nguyen, Thanh Hai Nguyen, and Renaud Sirdey. Improved compact formulations for a wide class of graph partitioning problems in sparse graphs. *Discrete Optimization*, 25:175–188, 2017. 5.3

[140] Jenny Nossack and Erwin Pesch. A branch-and-bound algorithm for the acyclic partitioning problem. *Computers & operations research*, 41:174–184, 2014. 2.4.1, 5.1

[141] Vitaly Osipov and Peter Sanders. n-level graph partitioning. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA), Liverpool, UK, September 6-8, 2010. Part I, 18*, pages 278–289. Springer, 2010. 2.3.3, 2.4.1

[142] Shih-Lian Ou and Massoud Pedram. Timing-driven placement based on partitioning with dynamic cut-net control. In *Proceedings of the 37th Annual Design Automation Conference*, pages 472–476, 2000. 2.4.1

[143] Shihliang Ou and Massoud Pedram. Timing-driven bipartitioning with replication using iterative quadratic programming. In *Proceedings of the ASP-DAC'99 Asia and South Pacific Design Automation Conference 1999 (Cat. No. 99EX198)*, pages 105–108. IEEE, 1999. 2.4.1

[144] Manfred W. Padberg and Giovanni Rinaldi. Optimization of a 532-city symmetric traveling salesman problem by branch and cut. *Operations Research Letters*, 6(1):1–7, 1987. 2.4.1

[145] Peichen Pan, Arvind K. Karandikar, and Chung Laung Liu. Optimal clock period clustering for sequential circuits with retiming. *IEEE transactions on computer-aided design of integrated circuits and systems*, 17(6):489–498, 1998. 2.4.1, 4.1.3

[146] David Papa and Igor Markov. Hypergraph Partitioning and Clustering. *Handbook of Approximation Algorithms and Metaheuristics*, May 2007. 6.1.4

[147] Ramakrishnan Pavithra Guru and Veeramuthu Vaithianathan. Ant colony optimization based partition model for VLSI physical design. In *Proceedings of 2020 International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–5, 2020. 2.4.2

[148] François Pellegrini. Graph partitioning based methods and tools for scientific computing. *Parallel Computing*, 23(1-2):153–164, 1997. 2.1.3

[149] François Pellegrini. Scotch and PT-Scotch Graph Partitioning Software: An Overview. In Olaf Schenk Uwe Naumann, editor, *Combinatorial Scientific Computing*, pages 373–406. Chapman and Hall/CRC, 2012. 2.1.3, 2.4.1, 4.4.2, 6.1.3

[150] François Pellegrini and Cédric Lachat. Process Mapping onto Complex Architectures and Partitions Thereof. Research Report RR-9135, Inria Bordeaux Sud-Ouest, December 2017. 2.1.3

[151] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *Proceedings of High-Performance Computing and Networking: International Conference and Exhibition HPCN EUROPE 1996 Brussels, Belgium, April 15–19, 1996 Proceedings 4*, pages 493–498. Springer, 1996. 1.3, 2.3.5

[152] François Pellegrini. Static mapping by dual recursive bipartitioning of process architecture graphs. In *Proceedings of IEEE Scalable High Performance Computing Conference*, pages 486–493, May 1994. 2.1.3, 2.3.5

[153] Merten Popp, Sebastian Schlag, Christian Schulz, and Daniel Seemaier. Multilevel acyclic hypergraph partitioning. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 1–15. SIAM, 2021. 2.4.1, 5.1

[154] Nicolas Pouillon and Alain Greiner. System on Chip Library Project. *[online] Available: https://largo.lip6.fr/trac/dsx/wiki*, 2010. 2.4.1

[155] Bryan T. Preas, Michael J. Lorenzetti, and Bryan D. Ackland. *Physical Design Automation of VLSI Systems*. Benjamin/Cummings Publishing Company, 1988. 1.3.1

[156] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76(3):036106, 2007. 5.1

[157] Rajmohan Rajaraman and Martin Derek F. Wong. Optimal clustering for delay minimization. In *Proceedings of the 30th international Design Automation Conference*, DAC '93, pages 309–314, New York, NY, USA, July 1993. Association for Computing Machinery. 2.4.1, 4.1.2

[158] Bernhard M. Riess, Konrad Doll, and Frank M. Johannes. Partitioning very large circuits using analytical placement techniques. In *Proceedings of the 31st annual Design Automation Conference*, pages 646–651, 1994. 2.3.1, 2.4.1

[159] Neil Robertson and Paul Seymour. Graph minors. I. Excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1):39–61, August 1983. 4.3.1, 4.3.2

[160] Julien Rodriguez, François Galea, François Pellegrini, and Lilia Zaourar. A hypergraph model and associated optimization strategies for path length-driven netlist partitioning. In *Proceedings of the 23rd International Conference on Computational Science (ICCS)*, pages 652–660. Springer, 2023. (document), 1.1.3, 1.2.1, 5, 6

[161] Julien Rodriguez, François Galea, François Pellegrini, and Lilia Zaourar. Path length-driven hypergraph partitioning: An integer programming approach. In Maria Ganzha, Leszek A. Maciaszek, Marcin Paprzycki, and Dominik Slezak, editors, *Proceedings of the 18th Conference on Computer*

Science and Intelligence Systems, FedCSIS 2023, Warsaw, Poland, September 17-20, 2023, volume 35 of *Annals of Computer Science and Information Systems*, pages 1119–1123, 2023. (document), 5, 5.5.1

[162] Julien Rodriguez, François Galea, François Pellegrini, and Lilia Zaourar. Hypergraph clustering with path-length awareness. In *Proceedings of the 24rd International Conference on Computational Science (ICCS)*. Springer, to appear. (document), 4

[163] Kalapi Roy-Neogi and Carl Sechen. Multiple FPGA partitioning with performance optimization. In *Proceedings of the 1995 ACM Third International Symposium on Field-programmable gate arrays*, pages 146–152, 1995. 2.4.1, 2.4.2

[164] Ro A. Rutman. An algorithm for placement of interconnected elements based on minimum wire length. In *Proceedings of the April 21-23, 1964, spring joint computer conference*, pages 477–491, 1964. 1.3.3

[165] Sadiq M. Sait, Aiman H. El-Maleh, and Raslan H. Al-Abaji. Evolutionary algorithms for VLSI multi-objective netlist partitioning. *Engineering Applications of Artificial Intelligence*, 19(3):257–268, April 2006. 2.4.2

[166] Sadiq M. Sait, Feras Chikh Oughali, and Mohammed Al-Asli. Design partitioning and layer assignment for 3D integrated circuits using tabu search and simulated annealing. *Journal of applied research and technology*, 14(1):67–76, 2016. 2.4.2

[167] Laura A. Sanchis. Multiple-way network partitioning with different cost functions. *IEEE Transactions on Computers*, 42(12):1500–1504, 1993. 2.4.2, 6.1.3

[168] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms. In *Proceedings of the European Symposium on algorithms (ESA)*, pages 469–480. Springer, 2011. 6

[169] Peter Sanders and Christian Schulz. Kahip v3. 00–karlsruhe high quality partitioning–user guide. *arXiv preprint arXiv:1311.1714*, 2013. 2.3.3, 2.4.1

[170] Dhiraj Sangwan, Seema Verma, and Rajesh Kumar. An efficient approach to VLSI circuit partitioning using evolutionary algorithms. In *Proceedings of 2014 International Conference on Computational Intelligence and Communication Networks*, pages 925–929, 2014. 2.4.2

[171] Gabrièle. Saucier, Daniel R. Brasen, and Jean-Pierre Hiol. Circuit Partitioning For FPGAs. In Gabrièle Saucier and Anne Mignotte, editors, *Logic and Architecture Synthesis: State-of-the-art and novel approaches*, IFIP Advances in Information and Communication Technology, pages 97–106. Springer US, Boston, MA, 1995. 5.2.3, 5.2.3, 5.2.3

[172] Garièle Saucier, Daniel R. Brasen, and Jean-Pierre Hiol. Partitioning with cone structures. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 236–239, Santa Clara, CA, USA, 1993. IEEE Comput. Soc. Press. 2.4.1, 5.2.3

[173] Sebastian Schlag. *High-Quality Hypergraph Partitioning*. Thesis dissertation, 2020. 2.3.3, 2.4.1, 2.4.1, 6.1.4, 6.2.1

[174] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. k-way hypergraph partitioning via n-level recursive bisection. In *Proceedings of the 18th Workshop on Algorithm Engineering and Experiments, (ALENEX 2016)*, pages 53–67, 2016. 2.3.3, 6.1.3

[175] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-quality hypergraph partitioning. *ACM J. Exp. Algorithmics*, mar 2022. 2.3.3

[176] Daniel G. Schweikert and Brian W. Kernighan. A proper model for the partitioning of electrical circuits. In *Proceedings of the 9th design automation workshop on Design automation - DAC '72*, pages 57–62, Not Known, 1972. ACM Press. 1.14, 1.3.3

[177] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-wesley professional, 2011. 3.1.2

[178] R Oguz Selvitopi, Ata Turk, and Cevdet Aykanat. Replicated partitioning for undirected hypergraphs. *Journal of Parallel and Distributed Computing*, 72(4):547–563, 2012. 2.3.2

[179] Minshine Shih, Ernest S. Kuh, and Ren-Song Tsay. Integer programming techniques for multiway system partitioning under timing and capacity constraints. In *Proceedings of 1993 European Conference on Design Automation with the European Event in ASIC Design*, pages 294–298. IEEE, 1993. 5.3

[180] Horst D. Simon and Shang-Hua Teng. How good is recursive bisection? *SIAM Journal on Scientific Computing*, 18(5):1436–1445, 1997. 2.1.1

[181] Adam Słowik and Michał Białko. Partitioning of VLSI circuits on subcircuits with minimal number of connections using evolutionary algorithm. In *Proceedings of the International Conference on Artificial Intelligence and Soft Computing*, pages 470–478. Springer, 2006. 2.4.2

[182] Kanagasabapathi Somasundaram. Multi-level sequential circuit partitioning for delay minimization of VLSI circuits. *Journal of Information and Computing Science*, 2(1):66–70, 2007. 2.4.1

[183] Yu-Hsuan Su, Emplus Huang, Hung-Hao Lai, and Yi-Cheng Zhao. Computer-Aided Design Contest Problem B: System-level FPGA Routing with Timing Division Multiplexing Technique. 3.3.2

[184] Subramanian Poothamkurissi Swaminathan, Pey-Chang Kent Lin, and Sunil P. Khatri. Timing aware partitioning for multi-FPGA based logic simulation using top-down selective hierarchy flattening. In *Proceedings of 2012 IEEE 30th International Conference on Computer Design (ICCD)*, pages 153–158. IEEE, 2012. 2.4.1

[185] Cliff Sze and Ting-Chi Wang. Optimal circuit clustering for delay minimization under a more general delay model. *Journal of Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 22:646–651, June 2003. 2.4.1, 4.1.2

[186] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975. 5.2.3, 5.2.3

[187] Katerina Tashkova, Peter Korošec, and Jurij Šilc. A distributed multilevel ant-colony algorithm for the multi-way graph partitioning. *International Journal of Bio-Inspired Computation*, 3(5):286–296, 2011. 2.4.2

[188] Mariem Turki, Habib Mehrez, Zied Marrakchi, and Mohamed Abid. Partitioning constraints and signal routing approach for multi-FPGA prototyping platform. In *2013 International symposium on system on chip (SoC)*, pages 1–4. IEEE, 2013. 2.4.1

[189] Eric S. H. Wong, Evangeline F. Y. Young, and Wai-Kei Mak. Clustering based acyclic multi-way partitioning. In *Proceedings of the 13th ACM Great Lakes symposium on VLSI*, pages 203–206, 2003. 5.1

[190] Honghua Yang and Martin Derek F. Wong. Circuit clustering for delay minimization under area and pin constraints. In *Proceedings of the European Design and Test Conference. ED&TC 1995*, pages 65–70, March 1995. 2.4.1, 4.1.2

[191] Zhengxi Yang, Zhipeng Jiang, Wenguo Yang, and Suixiang Gao. Balanced graph partitioning based on mixed 0-1 linear programming and iteration vertex relocation algorithm. *Journal of Combinatorial Optimization*, 45(5):121, 2023. 5.3

[192] Jih-Shyr Yih and Pinaki Mazumder. A neural network design for circuit partitioning. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pages 406–411, 1989. 1.3.3

[193] Dan Zheng, Xinshi Zang, and Martin Derek F. Wong. TopoPart: a Multilevel Topology-Driven Partitioning Framework for Multi-FPGA Systems. In *In proceedings of 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–8, November 2021. ISSN: 1558-2434. 2.3.4, 2.4.1, 3.2.4

[194] Xiaojin Zhu and Zoubin Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 15213, 07 2003. 5.1

[195] Konrad Zuse. Der plankalkül. 1972. 5.2.1

Publications

Conferences