Thermal-Aware Floorplan techniques for ASICs

Soumya Biswas, Albin Franzén



Lund University

Department of Electrical and Information Technology

in collaboration with Ericsson AB

Supervisor: Victor Åberg, Ali Zaher

Examiner: Pietro Andreani

June 10, 2026

Printed in Sweden
E-huset, Lund, 2026

Abstract

As transistors get smaller and smaller, the power density of an integrated circuit increases and thus the temperature increases. In an application specific integrated circuit (ASIC) the layout is tailor-made for each design, so the thermal signature of each component can be taken into account during the floorplanning process. Although approaches for thermal aware floorplanning exists, they aren't generalizable to new floorplans and can't handle hierarchical structures adequately. In this thesis we present a method to do thermal-aware compact floorplanning that is also generalizable through the use of reinforcement learning. We present a novel way to do hierachical divisions within the ASIC to improve connectivity, a new fast analytical thermal simulator and also utilize simulated annealing with reinforcement learning (SARL) in VLSI floorplanning for the first time. Our results are very successful demonstrating an automated flow for generating thermal aware floorplans.

I takt med att transistorerna blir mindre och mindre ökar effekttätheten i en integrerad krets, vilket leder till att temperaturen stiger. I en applikationsspecifik integrerad krets (ASIC) skräddarsys layouten för varje enskild konstruktion, vilket gör att varje komponents termiska signatur kan beaktas redan under planeringsprocessen. Det finns existerande lösningar för detta problem, men de kan inte generalisera till nya chip eller hanterar hierakiska strukturer. I denna avhandling presenterar vi en metod för termiskt medveten chip design som dessutom kan generaliseras genom användning av reinforcement learning. Vi presenterar ett nytt sätt att göra hierarkiska uppdelningar inom en ASIC för att förbättra konnektiviteten, har utvecklat en ny snabb analytisk termisk simulator och använder även reinforcement learning med simulerad annealing (SARL) tillsammans med parallell temperering för första gången. Våra resultat är mycket framgångsrika och demonstrerar en automatiserad metod för att designa ASIC med temperatur i åtanke.

Keywords: *Thermal-Aware Floorplanning, Reinforcement Learning, ASIC Design, Electronic Design Automation, Heat Dissipation, Placement Optimization, Physical Design*

Contents

Preface v
Acknowledgement . . . v
Contibutions . . . v
Popular Science Summary . . . vii
List of Figures . . . ix
List of Tables . . . xi
Key definitions and abbreviations . . . xiii

1 Introduction 1
1.1 Background . . . 1
1.2 Aim and delimitations . . . 2
1.3 Our contributions . . . 2
1.4 Thesis organization . . . 3

2 Literature review 5
2.1 VLSI floorplanning . . . 5
2.2 Clustering methods . . . 8
2.3 Thermal floorplanning solutions . . . 9

3 Methodology 11
3.1 Problem formulation . . . 11
3.2 Overview of solution strategy . . . 14
3.3 CB-trees . . . 15
3.4 Zone Division . . . 20
3.5 Clustering and quasi hierarchical states . . . 21
3.6 Directionality in tree generation . . . 23
3.7 Thermal simulation . . . 25
3.8 Optimizer . . . 27

4 Results 31
4.1 Custom dataset . . . 31
4.2 Thermal simulator . . . 31
4.3 SARL . . . 33
4.4 Tool usage . . . 38
4.5 Tradeoffs . . . 42

5 Discussion and Conclusion 49

5.1 Discussion 49
5.2 Conclusion 51

Appendix A **52**

Bibliography **55**

Preface

Acknowledgement

This master's thesis was commissioned by Ericsson AB, and we would like to express our sincere gratitude for the opportunity to carry out this work within their organization and for the resources and support they provided throughout the project. We are particularly grateful to our supervisor at Ericsson, Ali Zaher, whose mentorship and guidance was invaluable to us. We would also like to extend our thanks to our supervisor at LTH, Victor Åberg, for his feedback at various stages of our work.

Throughout the project we always came up with a quote of the day, and would like to share some of our favorite ones below.

- *"A bug is just pathfinder to the right fix."*
- *"When stuck in limbo, the path to reality may be hidden but never gone."*
- *"Creativity's a haze, delusions's a maze, true genius lies in chartering between the ways."*
- *"Reasonless conjecture is like a blind man's torchlight — everyone except him is aware of its futility."*

Contributions

This thesis was done by Soumya Biswas and Albin Franzén. The work was split evenly with Soumya being an electrical engineer taking more responsibility and work for the constraint handling to design viable ASICs, whereas Albin with a background in engineering mathematics worked more on the optimizer and thermal simulator. We spent approximately the same amount of work on the thesis and it has been an honor to work together.

We hereby affirm that this Master thesis was composed by ourselves, that the work herein is our own except where explicitly stated otherwise in the text. This work has not been submitted for any other degree or professional qualification except as specified; nor has it been published. Where other people's work has been used (either from a printed source, internet, or any other source), this has been carefully acknowledged and referenced. During the preparation of this thesis, we have used Generative AI and Large Language Models to assist us in coding. After using this tool/service, we have reviewed

and edited the content as needed and we take full responsibility for the content of the whole thesis.

Popular Science Summary

Over the past few decades decades, transistors, the smallest component on a chip, have been halving in size every 2 years. This decrease in size causes more current to be focused in a smaller area and thus the temperature increases. Higher temperatures have a significant impact on the performance of these chips, often called integrated circuits, leading to higher power consumption and reducing the lifespan of the circuit. It is therefore imperative to reduce the temperature of an integrated circuit, especially in situations where active cooling isn't available.

To suit user defined purposes, Application-Specific Integrated Circuits (ASICs) are designed with very specified functionalities and require tailor-made layouts for each application. Each ASIC can be broken down hierarchically into smaller blocks (in our case rectangles), where each block realizes a specific function. An ASIC layout is often referred to as a floorplan, and floorplanning is the process of choosing the best placement of blocks on a die given certain costs and constraints. Traditional floorplanning methods have primarily focused on increasing area utilization and decreasing wirelength, however as transistors have gotten smaller, the higher temperatures necessitate for it to also be optimized for.

In our thesis we developed an automated method for thermal-aware ASIC floorplanning that takes area, wirelength and temperature into account. Our algorithm first analyzes the connectivity between blocks in the floorplan and clusters them into groups. These groups are divided into larger zones, with connection cells between them to wire the floorplan together. Once the groups and zones have been set, a random floorplan is created using a representation called CB trees. This representation is iteratively perturbed such as moving a block, rotating a block or changing the aspect ratio. We use a simulated annealing acceptance criteria such that if a perturbation improves the overall cost then it is kept, and otherwise it is kept with a diminishing probability over the number of iterations. Commonly, simulated annealing is applied using random moves, however we designed an AI algorithm using a method called reinforcement learning to that has learned to make moves smartly from other ASICs. Once an approximate solution has been found, the solution goes through a refinement step to modify the aspect ratios of the blocks and flips in order to find the absolute best solution. For all of this to work we also had to develop our own thermal simulation software, and make it fast in order to run many iterations. Our final results have been very successful demonstrating an automated flow for generating thermal aware floorplans.

List of Figures

3.1 Valid floorplan (left) and invalid floorplan (right) due to overlap 11
3.2 Valid floorplan (left) and invalid floorplan (right) due to module A having different rotations and module B having different aspect ratios 12
3.3 Valid floorplan (left) and invalid floorplan (right) due to group A directly touch group B from different directions. 12
3.4 A valid floorplan containing a preplaced block (A), two right boundary blocks (B), and a hierarchical block (C) whose size is determined by two instances of C1 and one instance of C2. 13
3.5 Example of a floorplanning algorithm combining various blocks evaluated on a cost 13
3.6 Top level overview of how the algorithm works 15
3.7 Example of a B* tree and its packing. The anchor node is always in the bottom left 16
3.8 Example of a CB-tree and its packing. 17
3.9 Contour plane traversal and placement in CB-tree packing scheme. . . . 18
3.10 Contour plane traversal and placement with overlap checking. 18
3.11 The random generation of a B* tree 19
3.12 Zone Divided CB-tree Packing structure 21
3.13 Connectivity graphs (a)before and (b)after clustering 22
3.14 Segment Structure for single module type 23
3.15 Segment structure cases for creation of left and right child in case of two segments 23
3.16 Any node in a segment cannot have same module type as left and right child 24
3.17 Segment structure cases when attached as child of another node 24
3.18 Segment structure cases when a node is attached as a child of the segment 25
3.19 Directional alignment of segments through flip after tree generation . . . 25
3.20 Visualization of contribution from center block gets mirrored to enforce Neumann conditions 26
3.21 Overview of the optimizer architecture 27

3.22 Policy network for SARL 28
3.23 Before and after the refinement procedure of doing flips and modifying aspect ratios. 30

4.1 Packing of the dummy dataset (a) and thermal simulations using HotSpot (b) and our simulation (c) 32
4.2 Comparison of the difference in locations and temperatures for our simulation and HotSpot for the global max and the interior max of 1000 simulations 33
4.3 The training and validation curve evaluated on proprietary chip design using weighted negative log likehood. 35
4.4 Cost over per step of 1000 iterations for SA vs SARL on custom dataset without thermal cost. Mean highlighted in bold and 95% confidence interval shaded. 36
4.5 T-test per step of 1000 iterations for SA vs SARL on custom dataset without thermal cost. 36
4.6 Cost over per step of 1000 iterations for SA vs SARL on custom dataset with thermal cost. Mean highlighted in bold and 95% confidence interval shaded. 37
4.7 T-test per step of 1000 iterations for SA vs SARL on custom dataset with thermal cost. 37
4.8 Single-zone floorplan (a) placement and (b) thermal map. 38
4.9 Connectivity graph from optimizer 39
4.10 Four-zone floorplan (a) placement and (b) thermal map. 39
4.11 Eight-zone floorplan (a) placement and (b) thermal map. 40
4.12 12a zone floorplan 1 (a) placement and (b) thermal map. 40
4.13 12a zone floorplan 2 (a) placement and (b) thermal map. 41
4.14 12b zone floorplan (a) placement and (b) thermal map. 41
4.15 Best area floorplan 42
4.16 Best wirelength floorplan 42
4.17 Best thermal floorplan 43
4.18 Ternary plot of how varying weights affect the area utilization. 44
4.19 Ternary plot of how varying weights affect the wirelength. 44
4.20 Ternary plot of how varying weights affect the temperature. 45
4.21 The 3D Pareto front such that no data point has better area, thermal and wirelength than any other point. 45
4.22 The 2D Pareto front between area and temperature such that no data point has better area and temperature than any other point. The blue dots are the best floorplan per weight combination (computed with previous section weights) over 12 runs. 46
4.23 The 2D Pareto front between wirelength and temperature such that no data point has better wirelength and temperature than any other point. . 46
4.24 The 2D Pareto front between area and wirelength such that no data point has better area and wirelength than any other point. 47

List of Tables

3.1 Entity types and their action space 28

4.1 HotSpot comparision results for 1000 simulations 33
4.2 Hyperparameters used for the policy network and training procedure. . . 34
4.3 SA vs SARL on proprietary chip 35
4.4 SA vs SARL on custom dataset 35
4.5 Average iteration speed for SA and SARL 37
4.6 SARL Optimization metrics of best floorplan for different zone divisions and 5 grouping suggestions 41
4.7 Pure Weight Comparison Metrics 43

A1 Results by weight configuration 53

Key definitions and abbreviations

ASIC: Application Specific Integrated circuits. These refer to integrated circuits designed and optimized for specified functionalities.
Binary tree: A hierarchical data structure used for different purposes. It is a branching structure comprising of nodes having at most two children.
B*-Tree: Refers to a method of representation of floorplans using a binary tree.
CB Tree: Corner stitching compliant B*-Tree. It refers to another method of floorplan representation combining the corner stitching policies with B*-Trees.
Clustering: The process of grouping nodes together in a graph.
Die: In ASICs, it refers to the block of semiconducting material on which a circuit is fabricated.
EDA: Electronic Design Automation. These refer to a category of software tools used in design of electronic circuits.
Floorplan: The scheme of laying out various blocks of an ASIC together on a die for fabrication.
IC: Integrated circuit. Integrated circuits refer to miniaturized and compact electronic circuits, formed by integration of various components like transistors, resistors and capacitors fabricated on a chip.
RL: Reinforcement Learning. Refers to a method of machine learning involving an autonomous agent that learns to make decisions through interaction with its environment.
SA: Simulated Annealing. A baseline metaheuristic optimization algorithm.
Thermal delta: Refers to the difference in peak temperature between non uniform and uniform power distribution across the die for an ASIC.
VLSI: Very Large Scale Integration. It refers to the process of building integrated circuits through combination of millions/billions of transistors onto a single chip.

Chapter 1
Introduction

Over the past few decades semiconductor technology has been advancing at a rapid pace; with the number of transistors on an integrated circuit approximately doubling every two years following Moore's law. As transistors have gotten smaller and smaller, the power density increases and thereby the temperature of the chip. A consequence of this increase in temperature is an increase in power consumption brought by increased leakage currents in the transistors. It is therefore of utmost importance to keep the temperature of a chip low to reduce its power usage. In this thesis we present an automated method of doing so through thermal aware floorplanning.

VLSI (very large scale integration) refers to the process of building integrated circuits by combinations of millions/billions of transistors onto a single chip. To suit user defined purposes, integrated circuits are often designed with very specified functionalities, enabling the creation of more efficient products. Each Application-Specific Integrated Circuit (ASIC) can be broken down hierarchically into smaller blocks, where each block realizes a specific function. Floorplanning refers to the process of choosing the best placement of these blocks on the die given certain costs and constraints. In traditional floorplanning methods, optimizations have primarily focused on increasing area utilization and decreasing wirelength. However, as transistor sizes have gotten smaller, the cost of higher temperatures has become more pronounced necessitating for it to also be optimized for.

1.1 Background

Electronic Design Automation (EDA) is the process of automatically designing layouts for integrated circuits and has existed for many years. Floorplanning tools are a type of EDA that aim to place blocks (often rectangular) on a die in order to minimize a cost function. If there are only a few blocks and a lot of space, blocks can be placed iteratively one at a time on coordinates where available space exists. This is called the macro-placement problem; however, this approach cannot handle layouts where many blocks needs to be densely packed, since overlaps will result. Rather, to handle such cases a topological representation is needed which represents a block's placement in relation to other blocks, often referred to as compact floorplanning. There exist many different topological representations, but only a few are efficient in both packing (the process of converting the representation to coordinates) and constraint handling. One of the most notable representation methods that can do this are CB-trees [1] which are based upon B*-trees [2] that represents the floorplan as a binary tree. Efficient packing is achieved using an algorithm that can perturb this tree and optimize according to a cost-function. Traditionally, cost has primarily focused on area and wirelength, however due to the

decreased transistor sizes now temperature also needs to be optimized for.

Floorplan optimization using topological representations is often done using metaheuristic algorithms, which treat the representation as a 'black box' rather than directly analyzing the representation structure. Although common metaheuristics like simulated annealing and genetic algorithms are effective at finding high quality solutions, they are inherently memoryless, requiring a complete restart for every new problem instance. To overcome this lack of generalizability, recent advancements in artificial intelligence have shifted the focus toward reinforcement learning (RL). Google produced the landmark results in the field by using reinforcement learning to solve the macro-placement problem with AlphaChip [3], Li et al. [4] and Yu et al. [5] solve the compact floorplanning problem with reinforcement learning and Duan et al. solve the thermal macro-placement problem using it [6]. However what is lacking in literature is a solution for using reinforcement learning for compact thermal floorplanning.

1.2 Aim and delimitations

The aim of our thesis is to be the first to create a methodology that can generate generalizable compact thermally aware floorplans. For a floorplan to be a valid solution it must not have any overlapping modules, while also satisfying a number of constraints depending on the specific design. In this thesis we take into account hierarchical modules, preplaced modules, soft modules and directional connections as our constraints. Once valid solutions can be represented, the next aim would be to generate the best solutions given a cost function that would depend on three factors: area utlization, wirelength, and thermal hotspot delta compared to a uniform power distribution.

Once an algorithm has been devised that can generate viable solutions to the thermal floorplanning problem, it should be able to generalize to new constraints and designs. Similar to using accumulated knowledge in human IC design, it should not need to retrain from scratch when slightly modifying the constraints so our aim would be for the algorithm to learn from previous training cycles. We can summarize our goal with the following research questions:

- **What techniques and principles can be used for ASIC floorplan design to minimize thermal congestion?**
- **How can ASIC floorplan design be automated to implement such techniques?**

In this thesis we focus on creating early-stage thermally aware floorplans before actual design, synthesis or implementation. Thus, we do not take into account detailed routing, timing, and synthesis of the floorplan which would be done in later steps. Our solutions are also built for compact floorplanning, so if the space isn't constrained then other methods should be used.

1.3 Our contributions

Our main contributions is producing the first results that utilize reinforcement learning for the compact thermal floorplanning problem rather than the thermal macro-placing problem. We present a novel fast thermal simulation method that doesn't require any offline runs before inference, for the first time utilize simulated annealing with reinforcement learning in VLSI floorplanning, and propose a new hierarchical division process to divide a floorplan into smaller hierarchies.

1.4 Thesis organization

The continuation of this thesis is as follows: Chapter 2 will present a literature review covering current state-of-the-art floorplanning and thermal-simulation approaches. Our proposed thermal floorplanning solution will then be presented in chapter 3. In chapter 4, we evaluate our proposed floorplanning approach and study how reinforcement learning helps to improve the result. We conclude with a combined discussion and conclusion in chapter 5.

Chapter 2
Literature review

In this section we present current solutions and related works for thermal-aware floorplanning. In section 2.1 we introduce classical floorplanning and in section 2.2 we look at thermal aware floorplanning methods.

2.1 VLSI floorplanning

Floorplanning is the process of determining the location of components on a chip to minimize constraints such as area and wirelength. The problem has existed ever since the first integrated circuits in the 1960s and the first microprocessors in 1970s, but with an increase in computational power the first automatic floorplanning solutions came in the 1980s such as those by Heller et al. in 1982 [7]. In section 2.1.1, we introduce various works that devised methods of representing such problems, and in section 2.1.2, we present the algorithms that use such representations to create valid floorplans.

2.1.1 Representation methods

A floorplan representation is a way to represent the placement of blocks on a given area. The representations can be divided into two main categories: absolute representation and topological representation [8]. The absolute representation is a list of coordinates such as x_1, x_2, y_1, y_2 that is combined with a list of properties for each block and the connections between them. This representation is used by most thermal and synthesis tools due to it's simplicity and assumption of valid inputs. However it is disadvantageous for the compact floorplanning problem, as overlapping blocks may result, unless complicated masking procedures are used. Topological representations can represent compact solutions by looking at the relative positions between blocks through graphs or directional orderings, however, finding the exact position of a block may become more challenging [9].

Floorplanning representations can be further divided into slicing and non-slicing floorplans [9]. Slicing floorplans are ones in which all blocks can be represented in a 2D plane without any gaps, and slices can iteratively be used to split the plane. The earliest works on floorplan representation by Heller et al. in 1982 used a slicing tree to represent the iterative slices where the leaf nodes are the final blocks [7]. This caused redundancies where different trees can represent the same slicing floorplan, so Sechen et al. in 1986 devised normalized polish expressions (NPE) with unique representation [10]. An extension to slicing floorplans are mosaic floorplans in which there are no gaps, but the grid doesn't necessarily need to be iteratively partitioned. The foundational representation for these floorplans are corner block lists (CBL) devised by Hong et al. in 2000 that store lists of which block each corner is connected to [11].

A non-slicing floorplan is one that allows gaps between the blocks. Murata et al. introduced sequence pairs in 1996, the first representation to always produce a feasible floorplan when gaps are present [12]. Sequence pairs use two seperate lists for example $(abc), (bca)$ such that the order of the lists determine the resulting floorplan. This representation is valid and quite simple, however due to the two lists this approach is computationally complex, originally $\mathcal{O}(n^2)$, but later reduced to $\mathcal{O}(n \log \log n)$ [13]. Around this time many other representations were created to improve the packing time such as BSG [14], O-trees [15] and improving upon this B*-trees by Chang et al. [2]. B*-trees do search and insertion in $\mathcal{O}(1)$ time, along with packing and deletion in $\mathcal{O}(n)$ which allowed for a new generation of fast optimization algorithms to follow. B*-trees are limited to a subcategory of non-stitching floorplans called LB-floorplans meaning that all the blocks are compacted towards the bottom left, limiting the result to relatively rectangular layouts. Transitive closure graphs (TCG) by Lin et al. in 2001 which was improved in 2002 with TCG-S aimed to expand the tree-based methods to general floorplans [9][16][17].

With the creation of all these various floorplan representations, the next step was to add constraints to them in order to make them usable for actual floorplans and not only theoretical ones. Many of the representations already included some constraints such as proximity constraints and boundary constraints, however they were insufficient as the number of constraints increased. One constraint that received much attention is the symmetry constraint, specifying that parts of the floorplan needs to be symmetric, which a real design requires due to easier manufacturing, modularity and scalability. For example, if a part of the design is packed and wired efficiently in one area, then this can be mirrored to other parts of the floorplan reducing design and verification time.

The first works on symmetrical designs came from Balasa et al. in 2000 where they used sequence pairs to create symmetry groups, however multiple defects were found with this method by Koda et al. in 2007 who modified it using linear programming causing high run times [18][19]. Symmetry was also extended to O-trees in 2000, B*-trees in 2004 and TCG-S in 2005 but these all contained the issue of $\mathcal{O}(n^2)$ packing time which was first resolved with HB*-trees by Lin et al. in 2009 [20][21][22][23]. The same year, Xiao et al. released a method with comparable results to Lin et al. that can handle 2D symmetries and in 2011 Xiao's extends it to also handle other constraints, however as it is based on sequence pairs, the packing time becomes slow as the number of blocks increase [24][25]. In 2011 and 2016, two floorplan representation methods that are based on HB* trees and capable of handling more complex constraints in linear time called CB-trees [1] and QB-trees [26] were introduced. CB-trees can handle symmetry, proximity, preplacement, varying aspect ratios, boundary constraints and minimum separation constraints all in linear time. QB-trees can also handle maximum separation constraints and range constraints, while reducing the time complexity from $\mathcal{O}(nk)$ to $\mathcal{O}(n)$ for preplacement, boundary and minimum separation where n is number of blocks and k is number of constraints. However, there is one major caveat with QB-trees. Iteratively splitting the floorplan into quadrants around preplaced blocks limits the flexibility in arranging the floorplan around these preplaced blocks. This is acceptable with few preplaced blocks, but for complex floorplans CB-trees are preferred as they are more robust, and missing constraints can be handled by adding soft penalties to the cost function such as by Young et al. [27].

2.1.2 Floorplanning algorithms

There are many different ways of representing floorplans, which has led to a variety of methods capable of generating good floorplans given certain criteria. The first automatic floorplan design algorithm was by Heller et al. in 1982 where they developed a mathe-

matical approach to map inter-block connectivity into a distance space to generate the ideal slicing structure. In 1985, Liu et al. was the first to use simulated annealing for non-slicing floorplanning. This later became the benchmark used when evaluating new algorithms [28][10]. Simulated annealing introduced neighborhood searching which is the process that allows a given solution to be slightly perturbed, for example by swapping two blocks or rotating one to generate a new solution. It works by first initiating with a certain weight S, and then evaluating the difference between the current cost and the cost to get ΔC for doing some action. If the cost is lower, keep the new move, but if it is greater then only keep the new move with a probability $P = e^{-\Delta C/S}$. The weight S is reduced for each iteration to explore fewer and fewer worse perturbations. Simulated annealing was used in almost all the floorplan representation papers released in the early 2000s, such as sequence pairs, BSG, B*-trees, TCG-S and more, and it is still used in modern floorplanning tools like PARQUET [12][14][2][17][29]. Although being popular approach, simulated annealing suffers from getting stuck in local minimas once the weight has decayed sufficiently. [30].

Two prominent metaheuristic algorithms for floorplanning are genetic algorithms (GA) and particle swarm optimization (PSO) [30]. After simulated annealing, genetic algorithms were the main metaheuristic algorithm to be explored for floorplanning due to it's ability of exploring a greater variety of potentially good options giving better results. It was used already in 1996 by Rebaudengo and Reorda [31] for slicing floorplans, and in 2000 by Nakaya et al. [32] for non-slicing floorplans using sequence pairs. Genetic algorithms work by randomly generating a population of random solutions, and then letting these solutions cross-over and mutate to generate a new improved generation. With enough iterations the best individuals in the generation will converge towards a good solution. PSO, on the other hand, works by having a swarm of various solutions that keep track of their best solution and the swarms' best solution. In the next iteration the individual then updates to either a random perturbation, the best individual floorplan or the best swarm floorplan. Sun et al. [33] used this approach for B*-trees and Singh et al. [34] used PSO with sequence pairs.

In the 2010s, another algorithm called reinforcement learning became very popular because, unlike GA and PSO, it can generalize and use knowledge from one solution into another [35]. It gained traction after some remarkable results in complex environment such as video games and robotics [36][37][38]. These methods were brought to VLSI design when, in 2020, Google released a landmark paper called AlphaChip where they were able to use reinforcement learning to learn from previous designs to improve speedup and placement on new chips [3]. Their solution aimed to solve the macro-placing problem, which consists of fewer blocks that don't need to be compact so they can be placed iteratively one at a time by the algorithm. Many improvements to the macro-placing problem have been done in the most recent years such as MaskPlace [39], ChipFormer [40], AutoDMP [41] and EfficientPlace [42] which all leverage RL for macro-placement. Although macro-placing algorithms can produce very good results for large scale floorplanning where wirelength is more important than area, they perform poorer on classical compact floorplans [4].

The first time that reinforcment learning was used for compact floorplans was in 2020 by He et al. who presented a deep q-learning (DQN) method using reinforcement learning with a sequence pair representation [43]. Their method was very similar to simulated annealing except for the decision making; an agent with knowledge about the history of the cost function and how other blocks are affected, was used to decide on the next step. In 2022, this approach was improved upon with GoodFloorplan by Xu et al. who proposed that rather than only using handcrafted inputs, a graph convolution network (GCN) could take any floorplan netlist, represent it as an adjacency matrix, and encode it

as a vector [44]. This would then allow the RL agent to view the current layout along with the other inputs. Additionally, rather than using DQN, GoodFloorplan uses an advantage actor critic model (A2C) which can be more stable and data-efficient. Another approach to RL for compact floorplans was done by Amini et al. with CBL-RL where they used a corner block list representation to sequentially build the floorplan [45]. For the input, they instead represent the floorplan as a hypergraph and used a new finding by Bai et al. to encode the hypergraph with a hyperGCN [46]. Additionally, they not only encode the state space but also the action space using a dot-product attention mechanism, a necessity if many blocks are used. An improvement to the hypergraph representation was done by Guan et al. in 2024, fusing hyperGCN with a deep transformer q-network (DTQN). This will put focus on modules having many connections and a large area to achieve a good result. They used sequence pairs, and it shows that perturbing an existing solution is better than generating a new solution from scratch each time since it can get stuck in a local minima if a poor decision is taken early. Finally, the current state of the art for reinforcement learning with VLSI floorplanning was presented at Neurips 2025 with an algorithm that combines reinforcement learning with evolutionary algorithms also using transformers [4].

2.2 Clustering methods

Clustering refers to the process of grouping nodes together in a graph. It is a highly beneficial tactic for dividing floorplans into smaller hierarchical groups of strongly connected instances. Kernighan Lin Algorithm [47] was used previously in circuit partitioning for balanced bisections. Fiduccia-Mattheyses Algorithm [48] refined the KL algorithm further to be faster and have the ability to handle hypergraphs. However this was still limited to just bisections. Further partitions could still be achieved through recursive runs. Multilevel partitioning was introduced by hMETIS [49] in the late 1990s, and became immensely popular in VLSI research for partitioning schemes in EDA tools. However, it still required the number of partitions to be specified in advance, which is not always known from a floorplanning context. Louvain clustering [50] solves this problem as it works on the basis of automatic community detection.

Louvain clustering is based on the principle of optimizing modularity of each node. Here modularity refers to a measure of density of links inside a community as compared to density of links between communities. It is a scalar value between -1 and 1. It is given as

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j) \tag{1}$$

Here A_{ij} represents the weight between nodes i and j. k_i and k_j are sum of weights of the edges connected to node i and j respectively. c_i and c_j refer to their respective communities, $m = \frac{1}{2} \sum_{ij} A_{ij}$. δ is the Kronecker delta function where $\delta(c_i, c_j)$ is equal to 1 if $c_i = c_j$, otherwise 0. Louvain clustering has two phases and works iteratively. Initially, each node represented an individual community. In the first phase, the best possible move is calculated for every node such that the gain in modularity is maximized. For a node i in community C, the modularity gain is calculated as

$$\Delta Q = \left[\frac{\Sigma_{in} + k_{i,in}}{2m} - \left(\frac{\Sigma_{tot} + k_i}{2m} \right)^2 \right] - \left[\frac{\Sigma_{in}}{2m} - \left(\frac{\Sigma_{tot}}{2m} \right)^2 - \left(\frac{k_i}{2m} \right)^2 \right] \tag{2}$$

where Σ_{in} is the sum of weights of the edges inside the community C, Σ_{tot} is the total sum of weights of edges for all nodes inside the community, k_i is the sum of weights of edges

connected to node i, and $k_{i,in}$ is the sum of edge weights of i within the community C. In the second phase, a new network is built with the newly formed groups as the nodes. The new weights for the edges are calculated through summation of the edge weights of internal nodes between the two communities. This process is iterated until a maximum modularity is reached. this occurs when no move can produce abetter modularity score.

The standard Louvain clustering method has a limitation where the desired community size cannot be defined. Smaller communities may be merged even if it is not desirable, or very small communities may result. To control this behaviour and make the algorithm more robust, a resolution parameter γ can be used [51] where

$$\Delta Q = \left[\frac{\Sigma_{in} + k_{i,in}}{2m} - \gamma\left(\frac{\Sigma_{tot} + k_i}{2m}\right)^2\right] - \left[\frac{\Sigma_{in}}{2m} - \gamma\left(\frac{\Sigma_{tot}}{2m}\right)^2 - \gamma\left(\frac{k_i}{2m}\right)^2\right] \quad (3)$$

With $\gamma = 1$, we get the standard Louvain algorithm. Decreasing the value of γ leads to smaller, finer communities while increasing it leads to larger, coarser communities.

2.3 Thermal floorplanning solutions

The impact of maximum temperature on performance and reliability of ASICs has been greatly studied and worked on over the decades. Poor temperature gradients not only increase the power consumption of chips due to leakage but also drastically increase the cost of cooling. Thermal performance of ASIC designs can be greatly improved by optimizing floorplans. The use of neural networks for solving thermal floorplanning problems for power electronics was proposed by Andrzej Kos [52] in 1993. Previous works in low power floorplanning have shown that a more uniform power density gradient leads to lower peak temperatures [53]. This similar ideology was adapted in 1998 [54] with the goal of reducing the maximum temperature for gate array placements introducing the concept of superimposing temperatures of blocks. The advent of Hostpot [55] in 2003 popularized the idea of using thermal RC network models over finite element simulations because of their faster speeds with relatively accurate results. This model was used along with newer floorplanning tools to come up with effective solutions for thermal aware floorplans. Hotfloorplanner [56], introduced in 2005, used Hotspot to bring temperature inputs into their cost function used when optimizing slicing floorplans.

Although Hotspot has been used as a baseline for many fast thermal simulators, attempts have been made on faster designs. In 2007 Han et al. use simulated annealing on slicing floorplans that computes a single thermal scalar based on the shared lengths between the m hottest blocks [57]. The shared length method is fast but requires a slicing floorplan and lacks the accuracy to detect multiple hotspots over a full floorplan. This model is also used by Chen et al. in 2015 for distributed thermal floorplanning over multiple systems [58]. One realization that started occuring in the 2010s is that a larger more accurate thermal simulator can be used to tune values for a smaller more efficient one. In 2013 Wu et al. precompute the thermal profile for each individual block storing them in a table, which are then superimposed get the final result. Lin et al. (2011) proposes using multiple lookup thermal profiles for different parts of the chip, and Chen et al. (2017) extends on this by computing the thermal profile at various locations of the chip and uses bilinear interpolation to get the profiles between. The lookup table approach is also used by Duan et al. in 2025 [6] where they use reinforcement learning for the thermal aware macro-placement problem. Other methods that have become popular for their speed are machine learning models that input a power map and output a temperature map [59][60]. Downsides with both the thermal lookup table and the neural networks are they require offline precomputation before the inference is run, and they

cannot generalize if the die-size, the block aspect-ratio, or similar changes. Additionally, they cannot guarantee that boundary conditions are met.

Chapter 3
Methodology

3.1 Problem formulation

The aim for the thesis is to create a a generalizable compact thermal floorplanning method, and in this section we formulate the problem more concretely. We begin by describing what defines a valid floorplan, before explaining what defines a good floorplan. A valid floorplan must adhere to the following constraints:

1. A floorplan contains a set of rectangular instances $B = \{b_0, ..., b_n\}$ with bottom-left coordinates (x_i, y_i) such that no blocks overlap. This is illustrated in Figure 3.1.

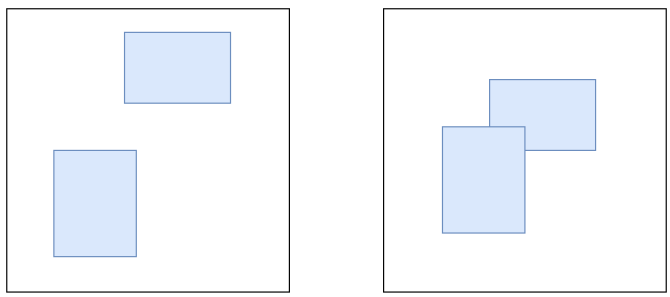


Figure 3.1: Valid floorplan (left) and invalid floorplan (right) due to overlap

2. The instances are grouped into modules $G = \{\{b_i, ...b_{i+k}\}...\}$ such that all instances of a module must have the same area, aspect ratio and be rotated the same direction, although mirroring is allowed. Figure 3.2 shows two cases of invalid blocks.

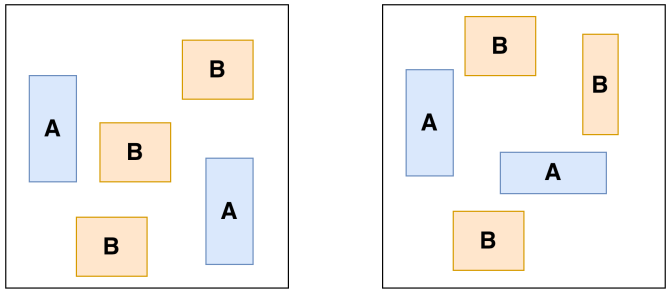


Figure 3.2: Valid floorplan (left) and invalid floorplan (right) due to module A having different rotations and module B having different aspect ratios

3. The aspect ratio for a given module is limited by $\frac{1}{l_i} \leq \frac{h_i}{w_i} \leq l_i$
4. All instances between two modules that have touching borders must touch from the same side, i.e. either they touch horizontal or vertical edges but not both in the same floorplan. Touching edges of both plane would need the different instances of the same modules to have different pin layout thereby requiring a different synthesis for each. Figure 3.3 shows the the cases for valid and invalid placements.

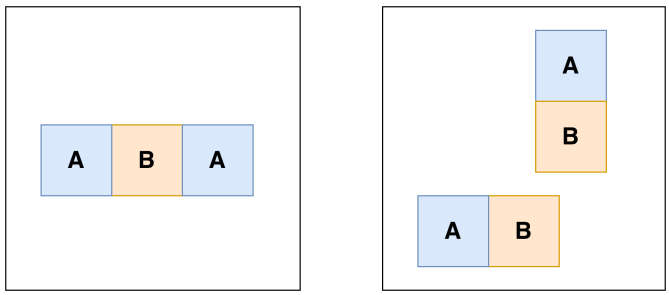


Figure 3.3: Valid floorplan (left) and invalid floorplan (right) due to group A directly touch group B from different directions.

5. Certain blocks $p_0, ..., p_1$ are preplaced(block A) and have fixed (x, y) coordinates, some must be placed on the boundaries(block b) and others create hierarchical modules(block C). Figure 3.4 demonstrates blocks belonging to all these categories.

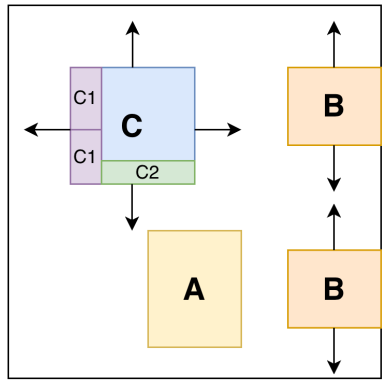
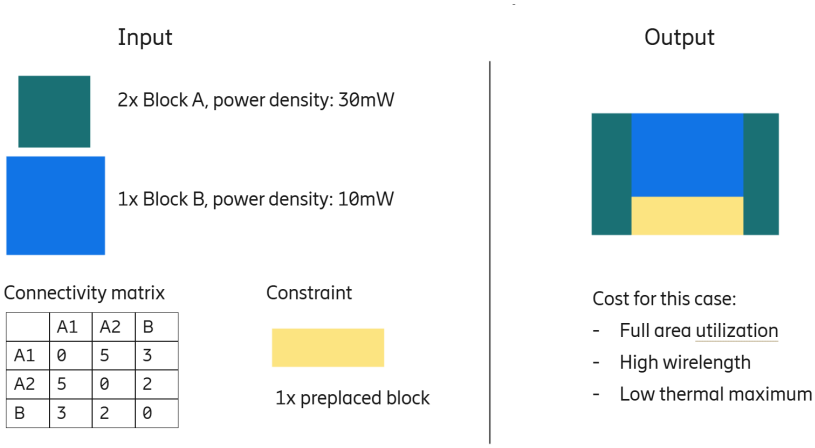


Figure 3.4: A valid floorplan containing a preplaced block (A), two right boundary blocks (B), and a hierarchical block (C) whose size is determined by two instances of C1 and one instance of C2.

The problem is to find a solution to a floorplan that minimizes or maximises certain properties. In this thesis we consider

1. The area utilization of the die which is a measure of the percentage of the die size area that is occupied by the blocks.
2. The total wirelength of connections between modules.
3. The thermal delta a floorplan which is a measure of difference of the peak temperatures for the actual power distribution and uniform power distribution across the die.

Additionally, the method used to find the solution should be generalizable meaning that it can learn from previous floorplan solutions to generate new ones in fewer iterations. This would allow for scaling and adaptivity for a wide range of constraints other than the ones mentioned.



	A1	A2	B
A1	0	5	3
A2	5	0	2
B	3	2	0

Figure 3.5: Example of a floorplanning algorithm combining various blocks evaluated on a cost

3.2 Overview of solution strategy

In this section we present an overview of our solution and some of the motivation for our solution strategy. The aim of our thesis is to create a floorplanning tool that can create a solution similar to the one shown in figure 3.5. This tool should take as inputs: a list of instances, a list of modules, a list of constraints, a connectivity matrix and zone positions (the zone positions determine the global layout of the ASIC). The outputs would be a valid floorplan according to the requirements presented in section 3.1. Our algorithm would be able to handle various combinations of these files and could produce different solutions depending on user specifications.

The first step for the tool is to parse all input files. The instance file contains the names of all the instances along with which module they belong to. The module file contains the area and power of the instances within the module. The constraint file contains a list of instances that have a certain constraint including their name, fixed x/y position if necessary, and the constraint type such as if it is a boundary module or a preplaced one. The connectivity matrix file contains a symmetric matrix with the number of connections between two instances, and the zone positions file is a list of how the floorplan should be divided into zones. To create a good floorplan, blocks with many shared connections are bundled together into hierarchical groups that will be placed together whenever possible. These groups may further be bundled into clusters, such that every zone packs a single cluster. Zones with same dimensions pack different instances of the same cluster (see section 3.4 for more details).

Once the floorplan has been divided into various zones containing clusters and groups, the next step is to place these entities onto the floorplan. Since we are dealing with compact layouts, this can not be done using just coordinates since it is likely that overlaps between blocks will result. Instead, each cluster and group is represented by a CB-tree. This topological representation is based on a binary tree where a right child indicates placement on top of the parent while a left child indicates that the child is placed to the right of the parent. The entire placement of the floorplan can be derived from this CB-tree representation, so it is sufficient to find a good binary tree representation in order to produce a good floorplan.

In order to find a CB-tree that produces the best floorplan, each binary tree, representing a cluster of group, first gets randomly populated before iterative perturbations optimize the placement. Perturbations are done using one of the following actions: moving a node in a tree, swapping two nodes, rotating one, or changing a node's aspect ratio. The decision of which move to take and which module, group or cluster to choose is taken by a reinforcement learning agent that has learnt from previous data which moves and entities produce the lowest cost. We use a simulated annealing acceptance criterion that accepts any moves that improves the cost, but worse moves are only accepted with a diminishing probability over time. Finally, the last step of the algorithm is to refine the solution which is done by ensuring that zones containing the same clusters are evenly symmetric (i.e. if there are 4 zones, there cannot be 3 with identical orientation), and then smaller aspect ratio modifications are performed. Through all of these steps a final floorplan is produced with low area, wirelength and temperature.

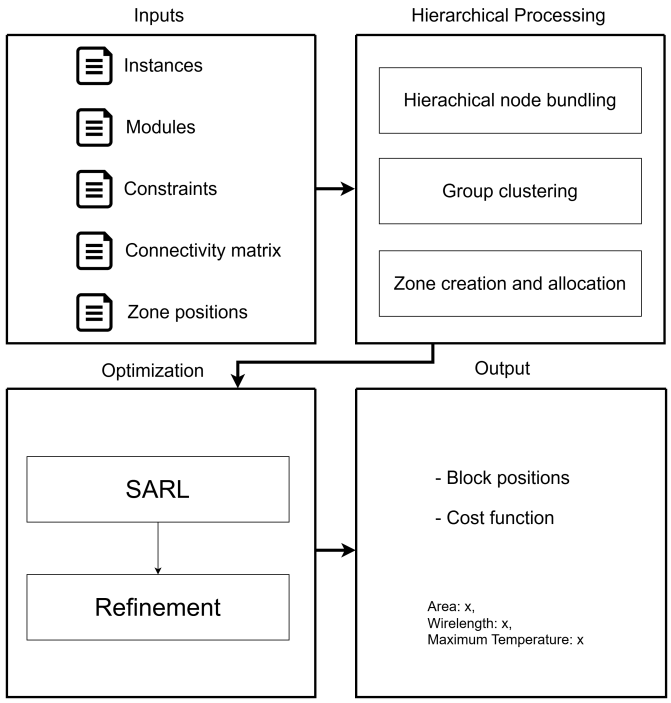


Figure 3.6: Top level overview of how the algorithm works

3.3 CB-trees

We decided to use corner stitching compliant B* trees (CB-trees) to represent our floorplans since B* trees can efficiently handle tree packing and constraints in linear time [1]. Figure 3.7 illustrates an example of B*-Tree representation and how it translates to a floorplan. CB-trees are built upon regular B* trees, of which the blocks are represented in a binary tree such that the left child of a node is packed to the right of the parent with $x_l = x_p + w_p$, and the right child of a node is packed on top of the parent such that $x_r = x_p$. The packing is done using preorder traversal which means that the left child is always packed first, and thus the packing will build up vertically which defines a contour line that can be used to keep track of the top-most y coordinate. Using this layout, packing can be done in linear time.

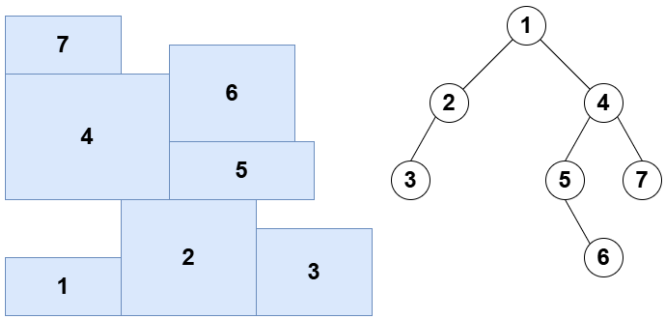


Figure 3.7: Example of a B* tree and its packing. The anchor node is always in the bottom left

The main issue with B* trees is that they can't handle constraints very well, for example if a preplaced module exists then when it is placed in the wrong location it must be moved to its proper spot, requiring a sorting and rearranging of the tree. To fix this, CB-trees introduce tiles and pointers such that each block defines a tile, each gap defines a tile and empty vertical contour tiles define the top most y-coordinates as shown in figure 3.8. The tiles are linked together using four pointers, *rt, tr, lb, bl* where *rt* is the rightmost top tile, *tr* is the topmost right tile, *lb* is the leftmost bottom tile and *bl* is the bottommost left tile. All the empty tiles are horizontally maximal in the sense that no empty tiles share a vertical edge. The addition of corner stitching to CB-trees allows for $\mathcal{O}(1)$ point searching to find the tile containing a certain point, $\mathcal{O}(1)$ area searching if a tile overlaps a certain rectangular region, and $\mathcal{O}(n)$ packing time. These search times are what allow CB-trees to efficiently and immediately check if a constraint is violated when a new block is placed, and use the pointers to move it to a new location.

3.3.1 Packing scheme

To pack a CB-tree, first all of the preplaced modules are placed and the corresponding tile plane is updated. Preplaced modules are packed only once, before the first optimization run. To simplify the initial tiling, preplaced blocks are sorted according to their y coordinate. Once the initial preplaced tiling plane has been created, the placement for the regular blocks follows the B* tree packing order, which packs the left child before the right one. The x and y coordinate of a block indicates its bottom left corner position while x_{end} and y_{end} refers to the top right corner.

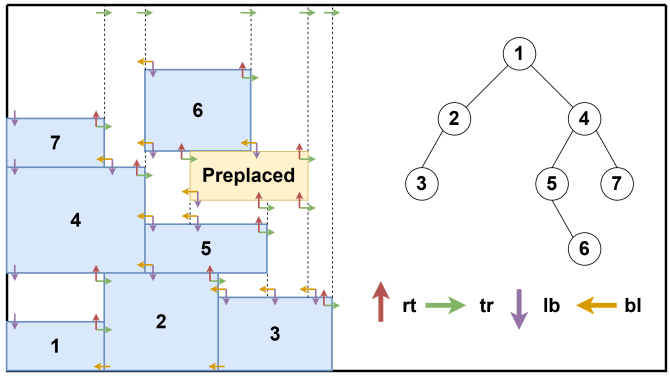


Figure 3.8: Example of a CB-tree and its packing.

To pack a node we begin by defining an initial placing tile which is the leftmost top tile of a parent for a right child (found through an *bl* traversal from *rt*) and *tr* for a left child. The x-location is known from the properties of B* trees to be $x_r = x_p$ or $x_l = x_p + w_p$, where x_p is the x coordinate of the parent node, w_p is the width of parent node, x_l is the x coordinate of a left child and x_r is the x coordinate of a right child. For the root placement, we start at the placing tile at starting location. Starting at this initial placing tile the y-coordinate is found through a contour tile traversal. The contour tile traversal goes to the tile pointed by *tr* if it is an empty tile and if the $x_{tr} < x_{pt} + w_n$, where x_{tr} is the x coordinate of the *tr* tile, x_{pt} is the x coordinate of initial placement tile and w_n is the width of the node. We traverse to the *lb* tile if current tile or the *lb* tile is a preplaced module(the yellow tile in figure 3.8). If there is a scenario where both the *tr* and *rt* tiles are non empty tiles or the current tile is a module(the blue tiles in figure 3.8) then it traverses to *rt* tile. Once a potential packing location is found, we check for overlaps with preplaced modules and if there is an overlap then a new placement tile is set at the *rt* of that preplaced module. From here a *bl* traversal is reached until a placing tile that contains the correct x-coordinate exists, along with a *rt* traversal until an empty tile is reached whose *rt* tile has a y coordinate above y_{end}. This determines the new packing location and restart contour traversal. During this *rt* traversal, we make sure to do a *bl* traversal at each step to reach the correct x-coordinate again. Every iteration of contour traversal after the preplaced block overlap check takes into account that we do not traverse down to an *lb* tile if it has a y coordinate below the preplaced tile's y_{end}. This process is repeated until all conditions are satisfied with no overlaps, and then the block is placed and the tiles are updated. For the packing location of the root node the user can determine a minimum x and y location and the tile is found by doing a *tr* traversal followed by an *lb* or *rt* traversal with similar backtracking to ensure correct coordinates, otherwise it is set to 0. This minimum x will automatically be enforced during the packing procedure, but to enforce the minimum y then we do not allow the contour tile traversal to go below the set minimum y-level.

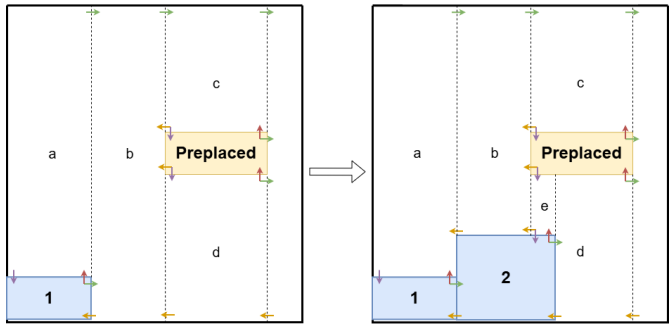


Figure 3.9: Contour plane traversal and placement in CB-tree packing scheme.

Following the example of the tree in Figure 3.8, we can see how the contour traversal logic follows in Figure 3.9 for node 2. Since it is a left child of node 1, the initial placing tile is determined as empty tile b. Now following the contour traversal logic, we move from b to c as pointed tr pointer of b. At tile c, we see $x_{tr} > x_{pt} + w_n$ but the lb pointer points to the preplaced tile. Thus we traverse down to the preplaced tile, traversing down again to reach tile d. At this tile we see all conditions are satisfied and we reach the end of our traversal. There are no overlaps and thus the node is placed, empty tile d is created, and all tile pointers are updated accordingly.

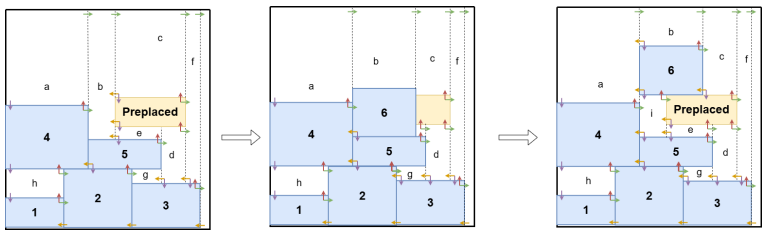


Figure 3.10: Contour plane traversal and placement with overlap checking.

We will now take a look at an example where a node would overlap with a preplaced block when packing node 6. Following the example in figure 3.10, the initial traversal starts from b, moves to c, and then down to the preplaced block before finally ending up in e where the traversal ends. With this placement, an overlap with the pre-placed block will result. Thus we begin our traversal again, determining the placing tile as b again, traversing to c. But now we do not traverse down further as the current tile would go below y_{end} of the preplaced module. Instead the traversal ends here and all tiles and pointers are updated accordingly after placing node 6. Finally, to ensure that blocks with a constraint specifying a top/right boundary placement, these blocks are placed after all other blocks to account for the width and height of the packed design.

3.3.2 Tree generation

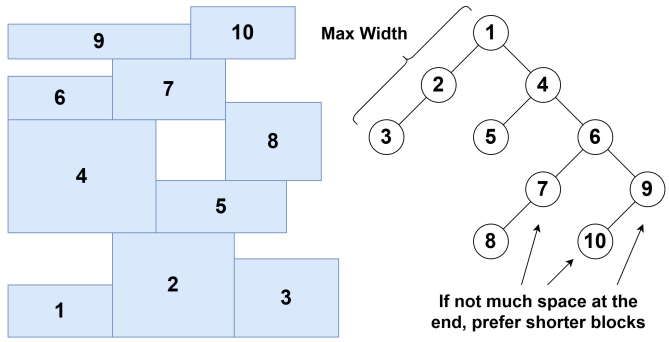


Figure 3.11: The random generation of a B* tree

We have now looked at the packing process. An important step that needs to take place before the packing is the initial binary tree generation. If a fully random CB-tree is used, a very tall tree would result at the B* packing algorithm moves nodes upwards if space does not exist. To enforce a maximum starting width we generate trees linearly, where nodes are packed as left childs until the maximum width is reached, and then create a new layer of left childs beginning as the right child of the first node of the previous layer as seen in Figure 3.11. This will ensure that the width of each layer is less than the maximum width and due to the packing procedure never moving nodes in the x-direction, the width is maintained during packing. In an attempt to limit the height, smaller blocks are with an increasing probability packed close to each other the closer we get to the upper boundary. The upper boundary is estimated from the total area of all nodes divided by the target placement density and the targeted width. The exact value isn't important due to this is only for generating initial floorplans, but around 0.75 takes a reasonable amount of deadspace into account. These considerations help in generating a good initial floorplan, thus requiring fewer iterations for optimization. Since the linear tree generation order is in the same order as the preorder packing, tree generation and packing can occur parallelly such that each time a random node is inserted in the tree, it can get packed immediately and the y-contour can be stored for those x-coordinates. This allows us to estimate the packing x and y for a new block based on the average block, which also gives an estimated distance to the boundary. If the packing location is far from the boundary then it is fully random, but if it is close to the boundary then smaller blocks are selected with higher probability. We use a softmax weighting scheme to determine the probabilities as

$$b_i \mid d \sim P(b_i \mid d) = \begin{cases} \dfrac{1}{n}, & \text{uniform, if } d < d_c \\ \dfrac{e^{-d^\alpha h_i/S}}{\sum_j^n e^{-d^\alpha h_j/S}}, & \text{softmax, if } d > d_c \end{cases} \tag{1}$$

where b_i is the i-th block, d is the progress between 0 and 1 towards the estimated upper boundary that is clamped to 1, and α is the progress factor that determines how much the progress should affect the distribution. The height of the block is given by h_i which is normalized by the maximum remaining block height and S is the spread

factor that determines the spread of the distribution especially once the progress kicks in. Determining the correct parameters can be a bit challenging, but d_c should be chosen so that there is still some space at the top to make changes such as selecting the d_c = estimated top y $- 2 \cdot$ max height or d_c = estimated top y/2. Using these guidelines of choosing d_c quite near to the boundary will lead to almost no increase in computation speed for the weighting since there are so few blocks left. Next there is a tradeoff between α and S, where α determines how quickly the border starts becoming important such as if $\alpha \to 0$ then the border is always important and if $\alpha \to \infty$ the border is important only at the very end. Reasonable value for diverse floorplans is in the range $0.5 < \alpha < 3$ as otherwise all the small blocks are clumped at the beginning or the end. Finally the parameter S determines the spread of the distribution with a greater spread causing more randomness. A good value for this can be found by fixing α, inspecting the average area utilization of generated floorplans and comparing to how random the floorplans look. If the floorplans generate many clumps of the same blocks then this would perform poorer for the optimizer since it lacks diversity, but would cause better area utilization short term. For future works, a metric for diversity could be computed and used for finding S automatically.

3.4 Zone Division

In most ASICs, the use of connect cells and buffers is prevalent. They are a derivative based on placement of other blocks and the connection and driving strengths required by them. Thus, they are not direct inputs from the user, rather, they are products of the floorplan itself. In certain use cases, floorplans without the connect cells might be unusable and thus futile. Traditionally, these connect cells are placed while the floorplan is built. We have here chosen to take an alternative approach by allocating space for these connect cells. However, placement of connects directly as preplaced blocks would somewhat hinder the potential for growth of the floorplan and restrict its packing possibilities. To overcome this issue, we introduced a zone division scheme where the floorplan is divided into smaller subsections. The division is done based on the initial die size and input in the form of coordinate values (x, x_{end}, y, and y_{end}). These values are initial bounds of the zone and they are updated as part of the packing process. Blocks are placed in zones based on their connectivity to other blocks. Once a block is placed in a zone, it cannot be moved to another zone. To reduce the impact of the zone division, the tool performs optimizations on the 5 best block divisions suggested based on their connectivity. We choose 5 suggestions in our current approach as we have seen it gives a good balance of exploring more solutions without increasing the computation time significantly. Each zone is treated as its own subfloorplan and gets it own inner CB-tree. The zones are packed sequentially to make sure that interconnections between the tiles in different zones are kept intact, and to make sure that pre-placed blocks are properly handled. With zones, the user gets more control over the floorplan structure through the zone placement and area allocation. Zone alignment is applied during packing where zones with same initial x, x_{end}, y or y_{end} maintain their equalities even if the actual value changes from the initial value. The initial area allocation affects the blocklist partitioning for zones which divides the blocks on the basis of ratio of the areas of the zones. Figure 3.20 illustrates an example of zone division. It depicts how each zone is packed based on it own CB-tree but share the same tile plane.

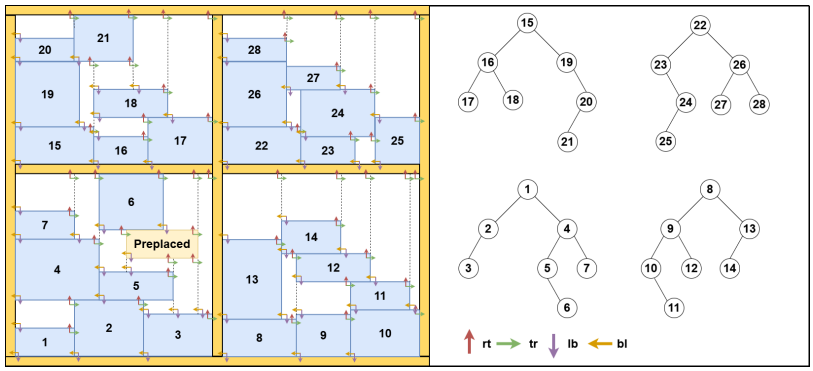


Figure 3.12: Zone Divided CB-tree Packing structure

3.5 Clustering and quasi hierarchical states

In most floorplans, it is preferable to place connected blocks with many connections close to each other. It is also preferred if the floorplan repeats symmetric structures when possible. To achieve this, we create quasi hierarchical groups by applying Louvain clustering [50] on the connectivity matrix.

In our implementation the modified algorithm is used to enable the groups sizes to be tuned, as mentioned before in section 2. To facilitate design reusability, the hierarchical grouping is performed such that hierarchical similarities are kept. This not only ensures the same packing scheme for all identical groups, but also saves computational overhead as each hierarchical module only needs to be packed once. To ensure this formation, we reduce the instance level connectivity graph to a module level graph. For a module pair m_1, m_2, the edge weight between them is calculated as

$$w(m_1, m_2) = \frac{1}{2} total(m_1, m_2) \frac{n_1 + n_2}{n_1\, n_2} \tag{2}$$

$$total(m_1, m_2) = \sum_{i \in m_1,\, j \in m_2} A_{ij} \tag{3}$$

where $w(m_1, m_2)$ is the edge weight between the modules, $total(m_1, m_2)$ is the combined sum of all edge weights between all instances of m_1 and m_2. n_1, n_2 denote the number of instances of m_1 and m_2 respectively. Running the clustering algorithm on the module level graph, we get the module level groups. The number of instances i of a group is calculated as the greatest common divisor of the instance counts of all modules inside that group.

This gives us our base suggestion for grouping modules. In our design, the groups remain unchanged through all iterations. To further explore valid grouping compositions, we calculate affinity of each module in groups. For every group we calculate the ratio of average internal connections to average external connections. The average number of internal connections is calculated as the sum of weights of all edges within the group divided by the number of edges. The average number of external connections is calculated as the sum of weights of all edges extending out of the group divided by the number of edges. Based on this ratio, we can determine how strongly a group is bound. We set a threshold at this stage. All modules in a group with a ratio lower than the threshold are considered to be flexible and thus we are allowed to change their grouping. For each flexible module m, we want to know how strongly it connects to each base group G_i.

We take all instances of the module in the group, sum up their connection weights to all instances in the target base group, then divide by the number of instances in the target group. That gives the affinity of the module for given group. It is based on the average connection strength per instance which is a fair comparison across groups of different sizes. This process is repeated for all flexible modules to create a descending sorted list based on the affinity for each module. Next, a choice list is created for each module based on the affinity scoring. The base grouping always remains as the top suggestion in this list. In our current implementation the list is limited to 3 choices, including more choices greatly impacts the computation time. Once we have the choice list for each flexible module, we take a cartesian product of all the lists. Each combination of this product suggests one possible alternate group composition. These combinations are then scored based on their modularity and sorted in a descending order. The top 4 suggestions are then used along with base grouping suggestion for further steps. We presently use only 5 suggestions to save on computation time.

In the next stage, we groups are redistributed based on the zone list to form clusters. These clusters are created in a way such that each zone gets one cluster. We start by assessing the given zone distribution to identify symmetric zones. Two zones are said to be symmetric if they have the same width and the height. The idea here is to attempt to create identical instances of clusters for symmetric zones. This cluster hierarchy is formed by zone based area matching. The groups are sorted area descending and assigned to zones based on available area. The instance count of a group should match or be a multiple of number of repetitions of its assigned zone. The groups assigned to same zone form the hierarchical structure we call cluster. After initial cluster formation, we run Louvain again on the instance subgraphs of each zone's modules to ensure that the division results in a balanced connectivity. Symmetric clusters and hierarchical modules are treated the same way; they are only packed once. The figure 3.13 shows the structure of an unchanged connectivity graph and the same graph restructured after grouping and cluster division.

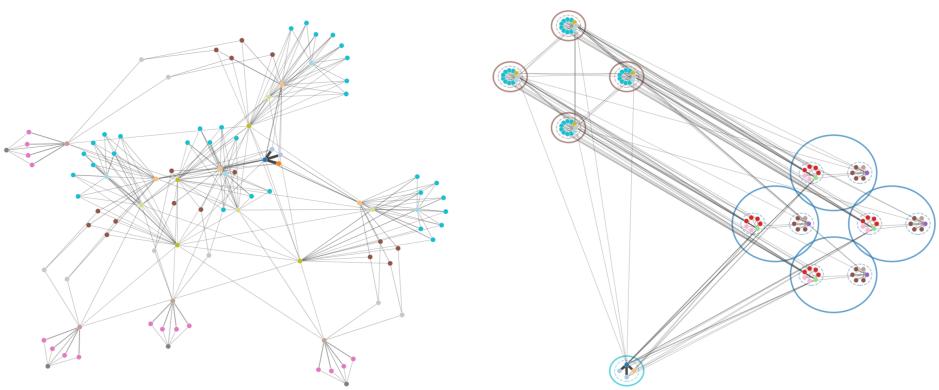


Figure 3.13: Connectivity graphs (a)before and (b)after clustering

Thus we create a nested hierarchical structure from initial instances, pairing them into groups, first based on their connectivity, and then further amalgamating them into clusters based on the zone division. In our implementation, this requires the user to smartly allocate zones to achieve a good floorplan. In practice, we have observed that starting off with a single zone lets the user study the composition of the floorplan before

trying out more complex zone division schemes in following runs. In general, grouping speeds up optimization by limiting our attainable solution space to more structured and logically sound floorplans.

3.6 Directionality in tree generation

The placement and directivity of ports is essential to consider in order to create a valid floorplan. To keep track of the directionality of different instances of a block same and ensure symmetric packing of groups, we devised a methodology to generate directionally aware trees to handle these constraints. We derived a list of rules for these groups that ensure randomized generation of these trees while adhering to the symmetric placement and directional connection clauses.

1. **Segment Detection:** Detect and list all instances that share the same module type. These form one or two segments based on weighted randomization.
2. **Equal Division for Two Segments:** If two segments are formed, the instances are divided as equally as possible between the two segments.
3. **Mirrored Branching for Equal Segments:** When both segments contain an exactly equal number of instances for each module type, the same branching structure is applied to both segments.
4. **Single Module Type Segments:** When a segment contains instances of only one module type, all instances within it are arranged in a right-child-only chain pattern, as illustrated in Figure 3.14.

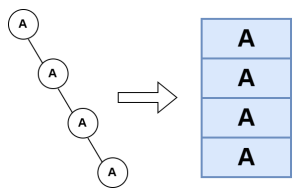


Figure 3.14: Segment Structure for single module type

5. **Left Child Rule for Multiple Module Types:** When a segment contains instances of more than one module type, a node may have another instance as its left child only if both share the same module type.
6. **Right Child Restriction:** Right child creation is permitted only within the primary right branch of the segment.

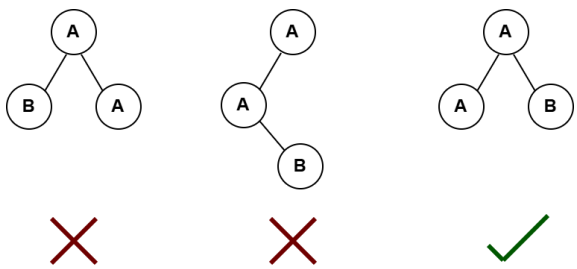


Figure 3.15: Segment structure cases for creation of left and right child in case of two segments

7. **Sibling Module Type Constraint:** At any level in the tree, a parent node cannot have a left child and a right child of the same module type simultaneously.

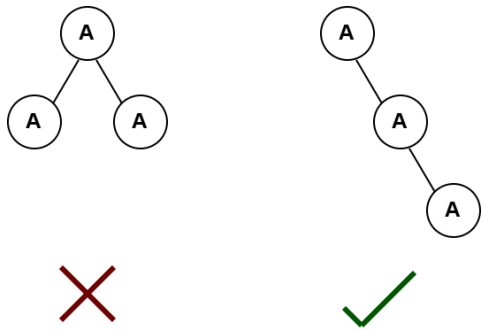


Figure 3.16: Any node in a segment cannot have same module type as left and right child

8. **Randomization with Segment Constraints:** The remaining instances outside the segments are randomized freely within the B*-tree, subject to the following two limitations on how segments are attached.
9. **Segment attachment as a child of another node:** A segment may only be attached to the rest of the tree as a left child of another block. The connection is made at the root node of the segment, as illustrated in Figure 3.17.

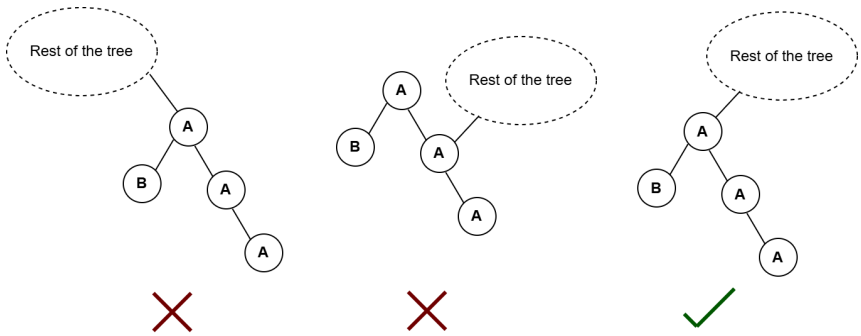


Figure 3.17: Segment structure cases when attached as child of another node

10. **Node attachment as a child of a segment:** A segment may only accept an external left child connection, and this connection must be made from the leftmost node of the segment, as illustrated in Figure 3.18.

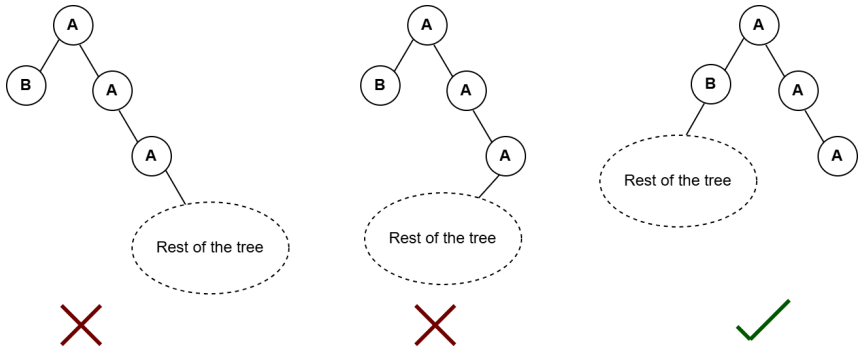


Figure 3.18: Segment structure cases when a node is attached as a child of the segment

11. **Post-Generation Directional Alignment:** After the full B*-tree is generated, in the case of two segments, it is checked whether these two segments are directly adjacent in the tree, i.e., one segment is the parent of the other. If this condition holds, the tree is left unchanged. Otherwise, one of the two segments is selected and all instances within it are flipped horizontally.

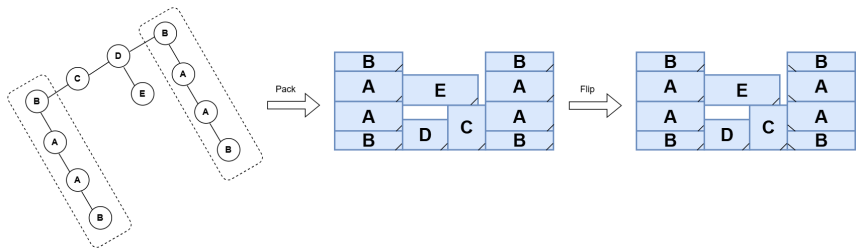


Figure 3.19: Directional alignment of segments through flip after tree generation

3.7 Thermal simulation

During the optimization process a fast method for determining thermal hotspots in the floorplan is needed. The most accurate solvers use a finite element method which scales at $\mathcal{O}(n^2)$ with n being the number of points in the simulation. In this section we present a method that can run in $\mathcal{O}(n)$ and can be parallelized making it independent on the number of points.

Our proposal to speed up the thermal simulation time builds upon the idea of using superposition for each block since that simplifies the problem by removing quadratic interaction terms. However, instead of creating a lookup table for each block, we derive a closed form analytical solution for the temperature dissipation of a block and superimpose all the blocks. To derive the thermal dissipation we begin from the 3D temperature distribution of a point source

$$T(r) = T_\infty + \frac{Q}{4\pi\kappa r} \tag{4}$$

Where $T(r)$ is the temperature from a given radius r from the source, T_∞ is the

ambient temperature, q is the power and κ is the thermal conductivity. To get the temperature distribution from a rectangle of point sources on the same 2D plane as the block, we integrate the point sources and set $z = 0$ to get

$$T(x,y) = T_\infty + \frac{q}{4\pi\kappa}\int_{y_1}^{y_2}\int_{x_1}^{x_2}\frac{1}{\sqrt{(x-x')^2+(y-y')^2}}\,dx'\,dy' \tag{5}$$

Note that in equation 5, q is the power density, and not the total power of the block. The integral of a uniform power distribution has a known analytical solution, as it also has an equivalent counterpart used for computing electrostatic potential [61]. The primitive function for this expression is defined as

$$F(u,v) = u\sinh^{-1}\left(\frac{v}{|u|}\right) + v\sinh^{-1}\left(\frac{u}{|v|}\right) \tag{6}$$

Where we can input the distances to the corners $\Delta x_1 = x - x_1$ to get the full solution

$$T(x,y) = T_\infty + \frac{q}{4\pi\kappa}\Big[F(\Delta x_1,\Delta y_1) - F(\Delta x_1,\Delta y_2) - F(\Delta x_2,\Delta y_1) + F(\Delta x_2,\Delta y_2)\Big] \tag{7}$$

For numerical stability we also add a very small number $\varepsilon = 10^{-12}$ to the denominator of $\sinh^{-1}$ to avoid division by 0. To speed up computations even further, for large distances we approximate the rectangle as a point with total power Q, giving the far field approximation

$$T(x,y) = \frac{Q}{4\pi\kappa\sqrt{\Delta x^2+\Delta y^2}},\ 0 << \sqrt{\Delta x^2+\Delta y^2} \tag{8}$$

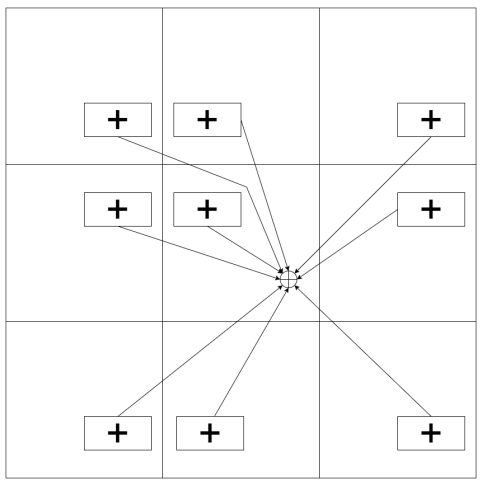


Figure 3.20: Visualization of contribution from center block gets mirrored to enforce Neumann conditions

The thermal conductivity moving off the ASIC is generally poor. Most heat will be dissipated through the top/bottom where we have the biggest surface area, only a small part will be dissipated on the sides. To simplify the analytical solution, we here assume that we have no heat dissipation on the sides which allows us to use Neumann conditions. This can be written as

$$\frac{dT}{d\mathbf{n}}(x, y) = 0,\ x, y \in \delta\Omega$$

where $\mathbf{n}$ denotes the normal vector to the boundary for a given point (x, y) on the boundary $\delta\Omega$. To enforce these conditions, we employ a common technique of mirroring [62], as seen in figure 3.20, where we mirror the floorplan along all 4 axes, and include their contributions when computing the temperature of a grid point as seen in figure 3.20. To handle the corners, we also add four doubly flipped mirrors outside of the floorplan corners. Using this technique, it is clear why the far field approximation in equation 8 is so beneficial, since the mirrors are placed far outside the floorplan and can almost always be computed as an inverse quadratic rather than using inverse hyperbolic functions.

3.8 Optimizer

In this section we present how the optimizer for finding good floorplans works. Figure 3.21 illustrates its general architecture.

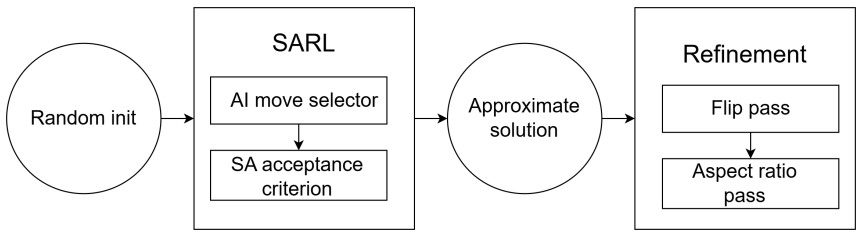


Figure 3.21: Overview of the optimizer architecture

Once a representation and a thermal simulator have been prepared, the next step is to optimize the floorplan. In order to do so we need to define a cost function that determines a metric for how good a floorplan is. In this thesis we consider area utilization, wirelength and maximum temperature, making the most straightforward cost function a linear combination of the three. The area utilization is the total area of all the blocks divided by their total bounding box, the wirelength uses the HPWL (half-perimeter wirelength) metric normalized by total connectivity in the connectivity matrix, and the thermal delta is given as a difference of peak temperatures compared to a floorplan where all the power was uniformly distributed over the die. The cost function is then

$$L = \alpha \cdot area + \beta \cdot HPWL + \gamma \cdot thermal \tag{9}$$

Where α, β, γ are parameters determining how much to weight area, wirelength and temperature.

In section 3.3-3.6 the representation method for a floorplan was presented and consists of modules, hierarchical blocks, groups and clusters and these are all packed into CB-trees at various hierarchical levels. It is possible to generate viable random floorplans by creating random trees, however the issue remains in how to generate such trees with the lowest cost. Perhaps the simplest solution to this problem is through local search, where beginning with a random solution it is randomly perturbed and if a perturbation improves the cost then it is kept else discarded. This naive search is prone to getting stuck in local optima, so instead of always discarding worsening moves, they can be accepted with a certain probability P. This method is called simulated annealing and in order for the algorithm to converge the worse move acceptance probability decreases over

time. The probability decay is determined by a simulated annealing parameter T, which determines the rate of decay of the probability $P(n) = e^{-n/T}$ where n is the iteration number.

In order to use simulated annealing an action space of perturbations must be defined. From our floorplan representation we have 4 different types of entities being modules, hierarchical blocks, groups and clusters. The action space for these entities consists of changing the aspect ratio of all instances of the same module, rotating a module/hierarchical block/group/cluster, flipping a module/hierachical block/group/cluster, swapping or moving within a cluster or hierarchical block, and repacking a group. By doing perturbations using these actions a floorplan with a good cost can be found.

Table 3.1: Entity types and their action space

Entity	Actions
Module	2 (rotate, change_aspect_ratio)
Hierarchical block	4 (rotate, swap, move, flip)
Group	3 (rotate, repack, flip)
Cluster	4 (rotate, swap, move, flip)

Although simulated annealing is a great method that produces good solutions, it lacks memory and generalization since it does not maintain knowledge from one solution into another. To incorporate prior knowledge into the algorithm, rather than letting simulated annealing take a random action each time, the probabilities of which action to take and which entity to modify can be learned by the policy network of a reinforcement learning agent that has been trained on other floorplans. The inputs to the policy network are a flattened list of modules, hierarchical blocks, groups and clusters, along with a state vector. The state vector includes the simulated annealing parameter along with all the costs, and the flattened list contains details about each entity specified in figure 3.22. The output of the policy network is a probability distribution of which entity to select, along with a probability distribution of which action type to select for that entity.

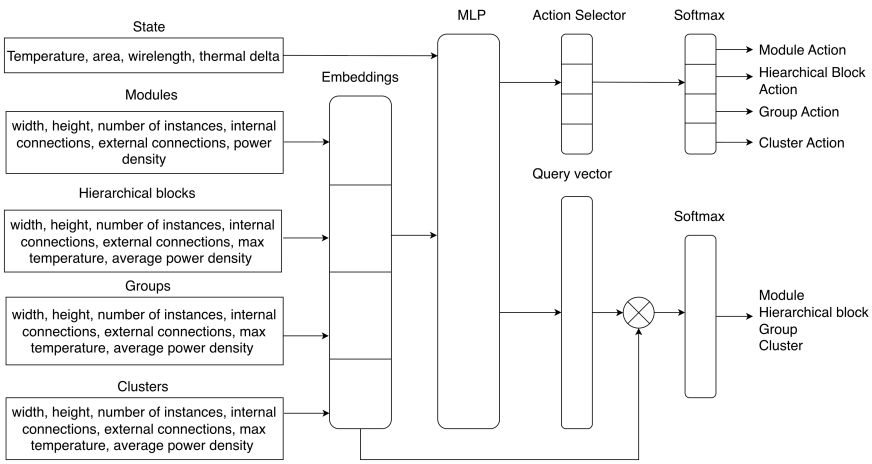


Figure 3.22: Policy network for SARL

Our reinforcement learning agent must be able to handle floorplans with varying numbers of blocks, so the input and output are mapped onto a dimension space with a size independent on the input/output size. The input embedding is done separately for each entity type with a linear layer that mean pools the features onto a fixed vector size and concatenates the embeddings of each entity type. This architecture allows for each entity type to have a different number of features. Next, the state and entity embeddings are fed into a multilayer perceptron (MLP) which has two output heads. The first head is the action selector which determines the probability of which action to take for each entity type after a softmax layer, and the second head is the entity selector. The entity selector produces a fixed-length query vector through a linear layer, that then takes the dot product with the input embedding of each entity. This result is fed through a softmax layer that converts the output to a probability distribution over the entities which can be used to select which entity to perturb. By being able to handle varying numbers of inputs and outputs it allows the model to generalize to new floorplans.

To produce viable results, the policy network must be trained on previous data. There exist multiple methods to train such a model, however due to the stochasticity of the action/cost landscape, there is hardly any signal how much a move will improve or worsen a certain state let alone the long term dependencies which makes classical reinforcement learning algorithms have a difficult time to converge. To ease convergence, our training is performed by comparing the result brought by different moves, all starting from the same initial state. Since the model will be used with a simulated annealing acceptance criterion we only care about improving moves since worsening moves are handled through the SA probability decay. For the training we then let simulated annealing run on the training data, if there is an improvement then we sample the entities and their respective actions and store the improving ones along with their cost.

The collected moves that cause improvements are used as soft supervision signals for training. Since we only care about relative improvements compared to the other improvements at that step, we normalize the improvement cost of each probed entity by dividing by the total cost improvement at the step to produce a weight $w_i = |\Delta C_i| / \sum_j |\Delta C_j|$, where ΔC_i is the cost reduction achieved by entity i. This weight describes how good a move is relative to the others at that step, making a good move heavily favored during training. In our simulated annealing algorithm, the entity to perturb is first randomly selected, then the action to be applied is randomly selected. To handle this joint probability we train using the weighted negative log-likelihood over both decisions jointly:

$$\mathcal{L} = -\frac{1}{N} \sum_{i \in \text{labels}} w_i \left[\log P(\text{entity}_i \mid s) + \log P(\text{action}_i \mid \text{entity}_i, s) \right] \tag{10}$$

where N is the number of labelled entity-action pairs in the batch. This is essentially the same as a cross-entropy loss where multiple targets are allowed per step, each weighted by their contribution to the improvement. The policy can be trained offline on a dataset of collected episodes using the Adam optimiser with gradient clipping, and a weight decay for better convergence. After training the policy network it can be used to replace the random entity selection of standard SA, increasing the probabiliy of choosing entities and actions that are likely to improve the current state while the SA acceptance criterion continues to handle worsening moves for exploration.

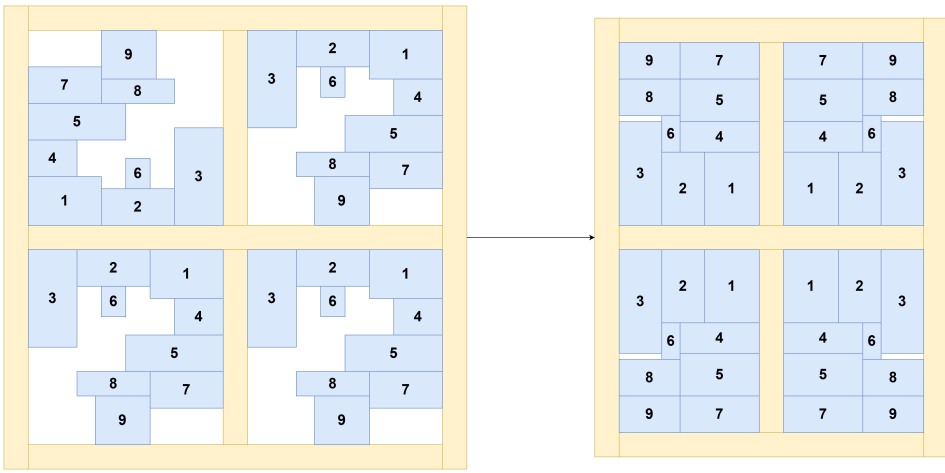


Figure 3.23: Before and after the refinement procedure of doing flips and modifying aspect ratios.

Once a viable floorplan has been found using SARL, the next step is to refine it (see figure 3.23. If there exist multiple zones with the same area and there is an even number of them, then we ensure that there exist an even number of horizontal and vertical flips of zones for a higher level of symmetry. So for example if 4 zones of a certain area exist, then either there are 0, 2 or 4 horizontal or vertical flips to avoid 3 zones facing one direction and 1 facing the other. After this, we do a flip pass to iterate through all the clusters and try each combination of valid flips through coordinate descent. Suppose our zones have a 4+4+1 structure, then the total number of combinations is $64+64+4 = 132$ which is a reasonable search space unlike a joint search. To ensure that the best local optimum is found this can be repeated multiple times. Once the best flip combination has been found the next step is to modify the aspect ratios of the blocks to perfectly align with each other. For each module, a decaying probe is done to modify the aspect ratio at $\pm\varepsilon$. If there is an improvement, then the change is accepted and the aspect ratio is changed to the new one. If there is no improvement then the step size is halved. This is repeated until $\varepsilon < 0.1\%$ of the current aspect ratio. Multiple aspect ratio passes can also be done since multiple modules depend on each other.

Chapter 4
Results

In this section we present the results of our work. We begin by presenting the custom dataset used throughout the results, how we use it to verify our thermal simulator, then compare SA with SARL, present various floorplan solutions and finally look at tradeoffs between area, wirelength and temperature.

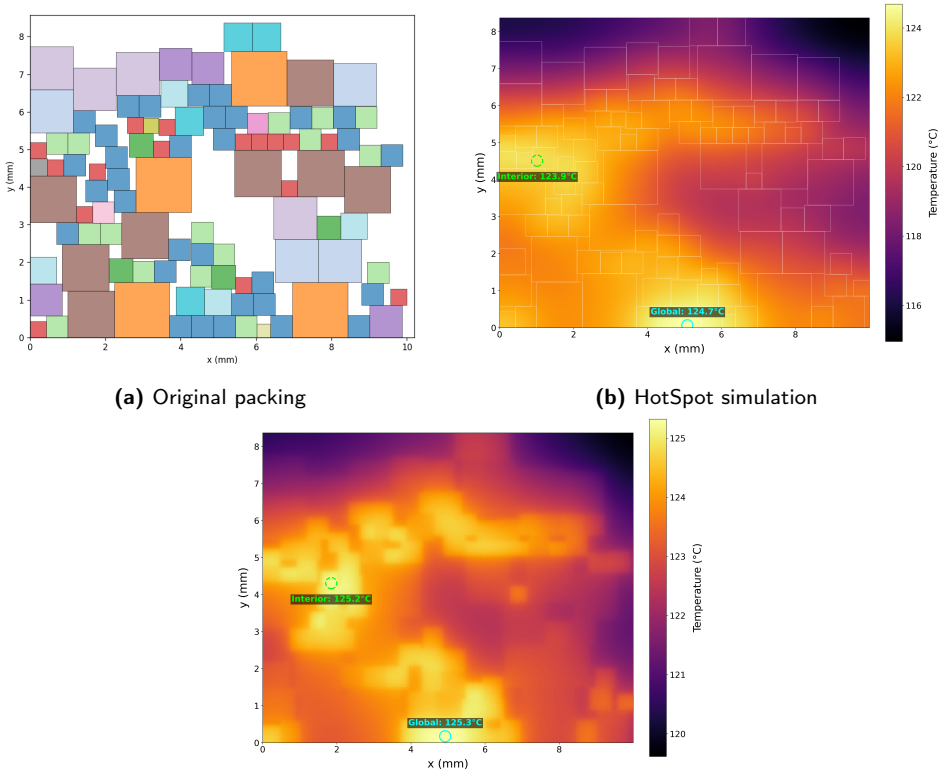
4.1 Custom dataset

All the results we present here are based on a custom dataset. Initially, we intended to use one of the open source benchmark datasets like GSRC [63] and MCNC [64]. However, most of these datasets lacked constraints and features which we had to take into consideration for our work. These limitations included lower instance counts for module types, fewer number of hierarchical modules and lack of scope for creation of multiple hierarchical instances of groups and clusters. To overcome these limitations, we created our own data set to suit our needs. We aimed to create a dataset with a lot of repeating structures having a complex connectivity to test out our clustering logic. We also chose to have varied power densities for blocks with several high power density blocks to test out our work at higher temperatures.

4.2 Thermal simulator

It is of high importance that the thermal simulator is accurate due to the thermal simulator having a significant effect on the final floorplan. We compare our thermal simulator with the open source tool HotSpot. We configured the parameters of HotSpot such as of die thickness, vertical convection ratio, conductivity and run on a single block to test it.

After configuration, we used the custom dataset and randomly packed it as a CB-tree. Since the most important feature of the temperature distribution for our simulation is the location of the maximum temperature we decide to use this as our metric. Due to the insulating boundary conditions on an ASIC most often this maximum temperature is near the boundary so we also compute the interior maximum that is within 1mm of the boundary. Figure 4.1 illustrates the comparison of heatmaps from hotspot and our thermal simulator for a random floorplan. We observe a similar temperature distribution for both, with the hotspot generated heatmap showing more blur.



(a) Original packing

(b) HotSpot simulation

(c) Our simulation

Figure 4.1: Packing of the dummy dataset (a) and thermal simulations using HotSpot (b) and our simulation (c)

We ran Hotspot and our thermal simulator for a 1000 different floorplans and compared the results to observe the overall variance. Table 4.1 shows our findings. It shows that our simulator has a mean global temperature error of 0.25°C and mean interior temperature error of -0.32°C. Since our optimizer is targeted for floorplans with very high peak temperatures, this margin of error is acceptable for it. The standard deviation for errors are also below 1°C which provides enough consistency for our use case. Figure 4.2 shows the distribution of data for the 1000 runs.

Table 4.1: HotSpot comparision results for 1000 simulations

Data Statistics	**Mean**	**Median**	**StdDev**
Global Max Temp (Own) °C	125.61	125.60	0.52
Global Max Temp (HotSpot) °C	125.86	125.75	1.39
Interior Max Temp (Own) °C	125.36	125.35	0.50
Interior Max Temp (Hotspot) °C	125.05	124.97	1.18
Differences	**Mean**	**Median**	**StdDev**
Global Pos. Distance (mm)	1.75	0.58	N/A
Interior Pos. Distance (mm)	1.62	0.58	N/A
Global Temp Error (°C)	0.25	0.16	0.97
Interior Temp Error (°C)	-0.32	-0.41	0.85

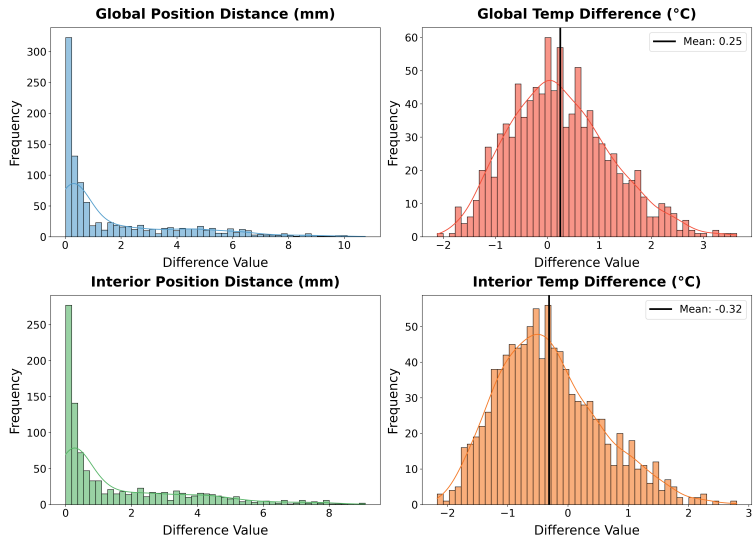


Figure 4.2: Comparison of the difference in locations and temperatures for our simulation and HotSpot for the global max and the interior max of 1000 simulations

4.3 SARL

In this section we present the training of SARL and compare it to SA. For our results we trained SARL on a single proprietary chip design which was also used for validation. We then evaluated the model on the custom dataset for testing. Table 4.2 lists the parameters used when training our model.

Table 4.2: Hyperparameters used for the policy network and training procedure.

Parameter	Value
Policy Network Architecture	
Entity embedding dimension	64
Query MLP hidden dimension	128
Module feature dimension	6
Hierarchical block feature dimension	7
Group feature dimension	7
Cluster feature dimension	7
Cost context dimension	4
Query MLP layers	2 (Linear $\rightarrow$ ReLU $\rightarrow$ Linear)
Query MLP input dimension	256 (4$\times$ embed_dim)
Activation function	ReLU
Action heads	4 (one per entity type)
Module actions	2 (rotate, change_aspect_ratio)
Hierarchical block actions	4 (rotate, swap, move, flip)
Group actions	3 (repack, flip, rotate)
Cluster actions	4 (flip, rotate, swap, move)
Training	
Algorithm	Imitation learning (behavioural cloning)
Optimiser	Adam
Learning rate	3×10^{-4}
Weight decay	1×10^{-4}
Batch size	4096
Epochs	10000
Total number of improving moves	331247
Validation split	10%
Episode Collection (SA)	
SA iterations per episode	100
Initial SA weight	0.05
Final SA weight	1×10^{-4}
Cooling schedule	Geometric
Training Cost Function Weights	
Area utilisation weight (w_{area})	0.45
HPWL weight (w_{hpwl})	0.10
Thermal weight (w_{thermal})	0.45

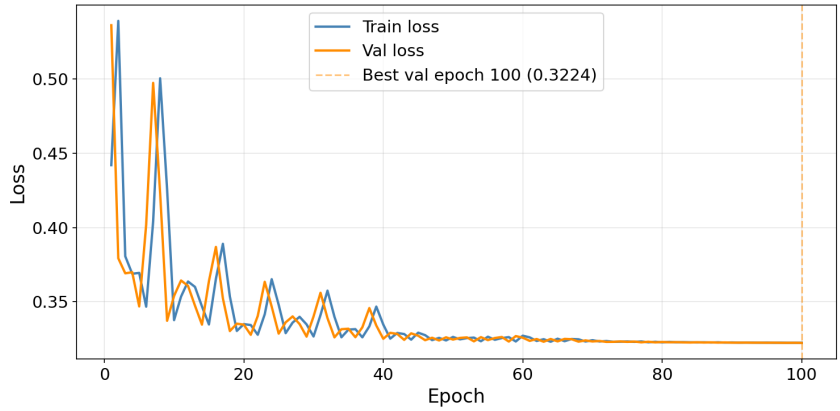


Figure 4.3: The training and validation curve evaluated on proprietary chip design using weighted negative log likehood.

The training of the SARL model was done through imitation learning explained in section 3.8 by running 1000 trajectories of simulated annealing on a proprietary chip design that ran for 10000 iterations each. Each of these trajectories contained one of 5 random groupings from the Louvain group suggestions, whereas the zone divisions remained the same for all runs. The validation data for the training was evaluated on unseen SA trajectories with one of the 5 groupings and the same zone division as the training data with the loss curves for the weighted negative log likelihood seen in Figure 4.3.

Once the training was completed, our SARL model was compared to SA by running both models 100 times with 1000 iterations. This was done for both the proprietary design and the unseen custom dataset which allows us to test if the model can generalize to new designs. For each iteration we compute the mean and standard deviation of the cost, and using this we can compare the distributions of SA vs SARL using a T-test assuming normal distribution. The p-value of the T-test determines how confident we are that the mean of SARL is greater than the mean of SA. We do this comparison for running SARL both with and without thermal simulations to test whether our model can generalize with a temperature cost, and the results are presented in Table 4.3 and 4.4.

Table 4.3: SA vs SARL on proprietary chip

Condition	Method	Sugg. 0	Sugg. 1	Sugg. 2	Sugg. 3	Sugg. 4
With Thermal	**SA**	0.150 ± 0.030	0.166 ± 0.025	0.166 ± 0.025	0.158 ± 0.027	0.184 ± 0.030
	SARL	$\mathbf{0.122} \pm 0.028$	$\mathbf{0.131} \pm 0.025$	$\mathbf{0.134} \pm 0.022$	$\mathbf{0.126} \pm 0.027$	$\mathbf{0.140} \pm 0.027$
Without Thermal	**SA**	0.208 ± 0.029	0.215 ± 0.024	0.217 ± 0.022	0.209 ± 0.024	0.233 ± 0.031
	SARL	$\mathbf{0.188} \pm 0.026$	$\mathbf{0.189} \pm 0.030$	$\mathbf{0.186} \pm 0.023$	$\mathbf{0.187} \pm 0.022$	$\mathbf{0.199} \pm 0.023$

Table 4.4: SA vs SARL on custom dataset

Condition	Method	Sugg. 0	Sugg. 1	Sugg. 2	Sugg. 3	Sugg. 4
Without Thermal	**SA**	0.155 ± 0.016	0.153 ± 0.026	0.138 ± 0.020	0.131 ± 0.023	0.126 ± 0.029
	SARL	$\mathbf{0.136} \pm 0.016$	$\mathbf{0.127} \pm 0.027$	$\mathbf{0.114} \pm 0.023$	$\mathbf{0.110} \pm 0.023$	$\mathbf{0.097} \pm 0.025$
With Thermal	**SA**	0.241 ± 0.012	0.232 ± 0.020	0.224 ± 0.020	0.224 ± 0.022	0.214 ± 0.025
	SARL	$\mathbf{0.226} \pm 0.014$	$\mathbf{0.211} \pm 0.018$	$\mathbf{0.206} \pm 0.021$	$\mathbf{0.210} \pm 0.021$	$\mathbf{0.195} \pm 0.024$

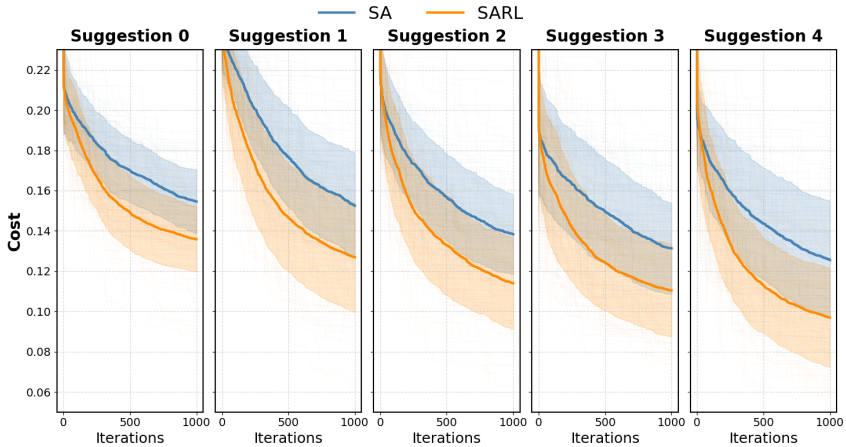


Figure 4.4: Cost over per step of 1000 iterations for SA vs SARL on custom dataset without thermal cost. Mean highlighted in bold and 95% confidence interval shaded.

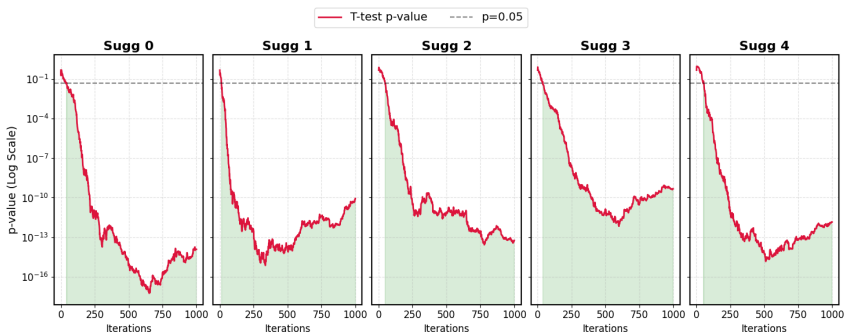


Figure 4.5: T-test per step of 1000 iterations for SA vs SARL on custom dataset without thermal cost.

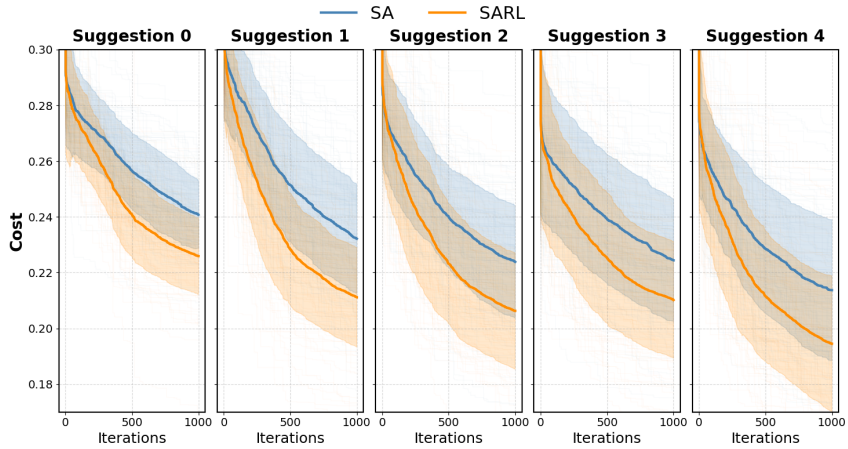


Figure 4.6: Cost over per step of 1000 iterations for SA vs SARL on custom dataset with thermal cost. Mean highlighted in bold and 95% confidence interval shaded.

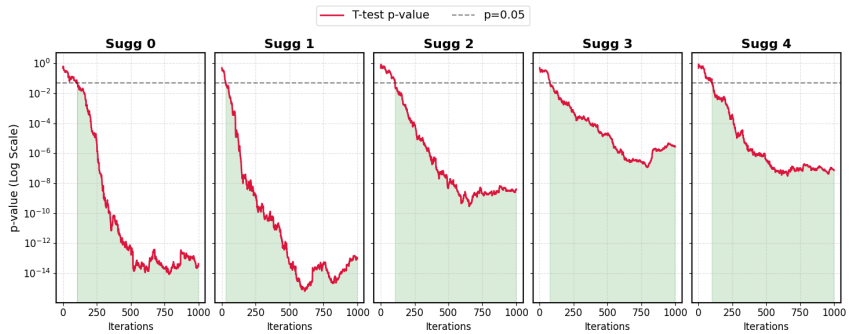


Figure 4.7: T-test per step of 1000 iterations for SA vs SARL on custom dataset with thermal cost.

Table 4.5: Average iteration speed for SA and SARL

Algorithm	Speed (it/s)	Latency (ms/it)
SA	189 ± 6	5.3 ± 0.2
SARL	130 ± 8	7.7 ± 0.5

From Table 4.3 and 4.4 we see that SARL provides floorplans with lower mean cost with and without temperature costs for all 5 suggestions. This is visualized in Figure 4.4 and 4.6 which show how the difference of the two methods vary over the number of iterations. The T-tests in Figure 4.5 and 4.7 show the p-values reaching far below $p = 0.05$ demonstrating that SARL has a significantly lower cost than SA.

4.4 Tool usage

In the section we present how our tool can be used to generate floorplans given a custom data set and no additional information about how the floorplan should be structured. The baseline for our comparison is 120 °C, that is when the floorplan has a uniform power density. We started the first run of the optimizer with only a single zone using the weights $w_{area} = 0.5, w_{hpwl} = 0.2, w_{thermal} = 0.3$. The single zones gives an idea about the sizes of different groups present and their number of repetitions and the weights prioritize area with a bit of temperature and wirelength. We also study the instance connectivity graph after Louvain clustering as this allows us to see not only how different groups are connected to each other, but also observe how closely different instances of same group connect. This helps us later in formulating the next zone division schemes. The optimizer also gives a rudimentary zone suggestion in form of how many distinct groups should be present along with their number of required repetitions.

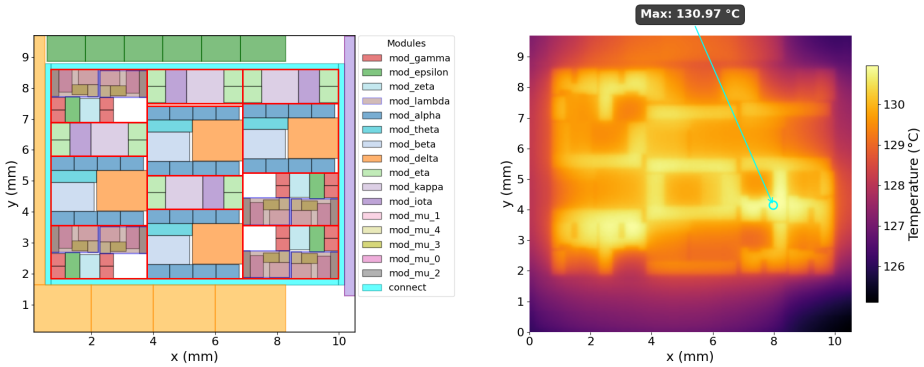


Figure 4.8: Single-zone floorplan (a) placement and (b) thermal map.

From the results of the first run in Figure 4.8 and its corresponding clustering in Figure 4.9, we observe 3 repeating groups, each with 4 repetitions. The best floorplan from this run is illustrated in Figure 4.8. Looking at the connectivity graph in Figure 4.9, we observe the inter and intra connectivity among groups and their instances. We can correlate the group sizes and their connectivity for deciding further zone division schemes.

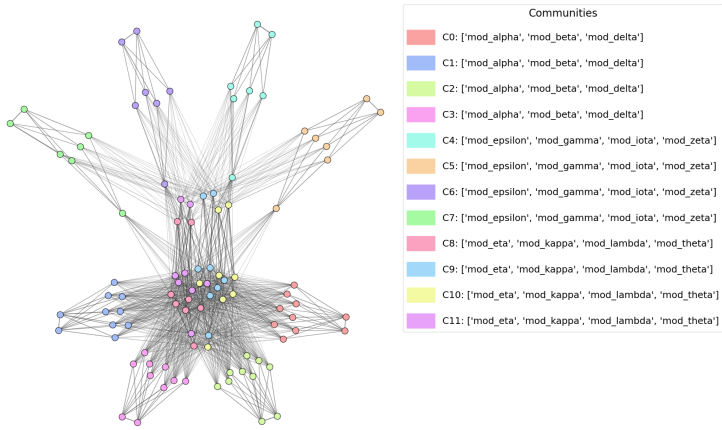


Figure 4.9: Connectivity graph from optimizer

Since we see all groups repeating 4 times, we try a zone division scheme of 4 symmetric zones. The best result for this configuration is given below in Figure 4.10. We see significant improvement in connectivity costs as well as reduction in thermal delta here compared to the first run as shown in table 4.6. We also see a floorplan with more reusable clusters.

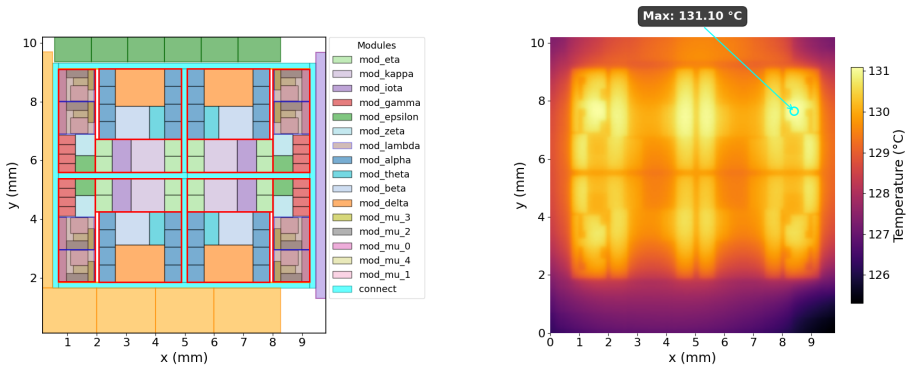


Figure 4.10: Four-zone floorplan (a) placement and (b) thermal map.

To test out how further clustering could affect the floorplans, we divide the floorplan into 8 zones with 2 distinct zone groups with 4 repetitions each. We retain the area information from previous runs and use them for zone area sizing in this zoning scheme. This allows us to maneuver which groups we want to be clustered and in which formation they will be placed.

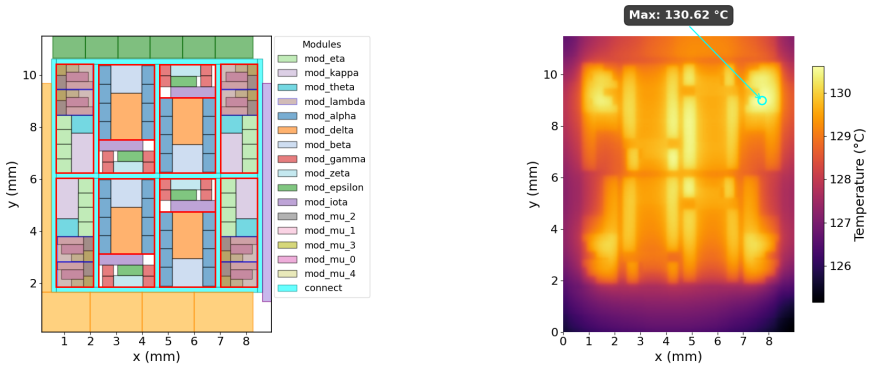


Figure 4.11: Eight-zone floorplan (a) placement and (b) thermal map.

Next we follow the zone suggestion scheme of the optimizer and implement a 12 zone scheme with 3 distinct zones each having 4 repetitions. We have tried out two alternate implementations of this based on how the zones are laid out. The best results of both runs are presented below in Figure 4.12, 4.13 and 4.14. It is interesting to note that for our first implementation, we get two different floorplans with equal overall cost but different individual costs, as shown in table 4.6, highlighting the tradeoffs between them.

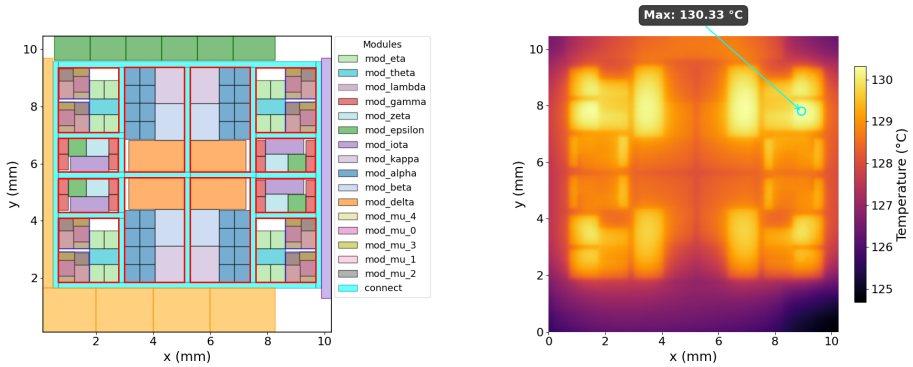


Figure 4.12: 12a zone floorplan 1 (a) placement and (b) thermal map.

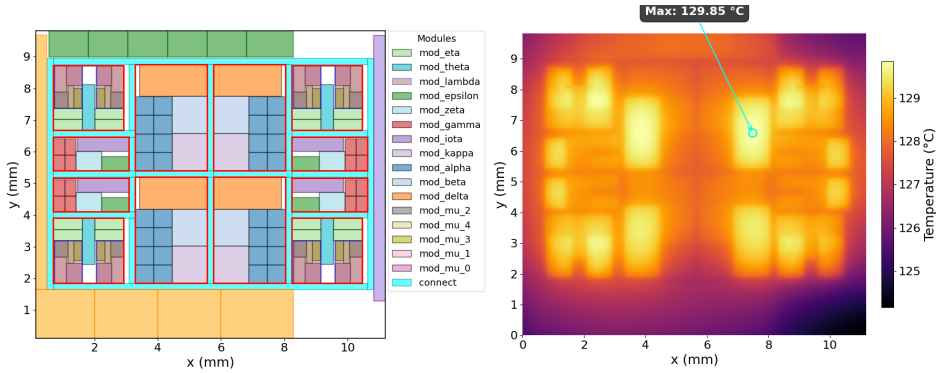


Figure 4.13: 12a zone floorplan 2 (a) placement and (b) thermal map.

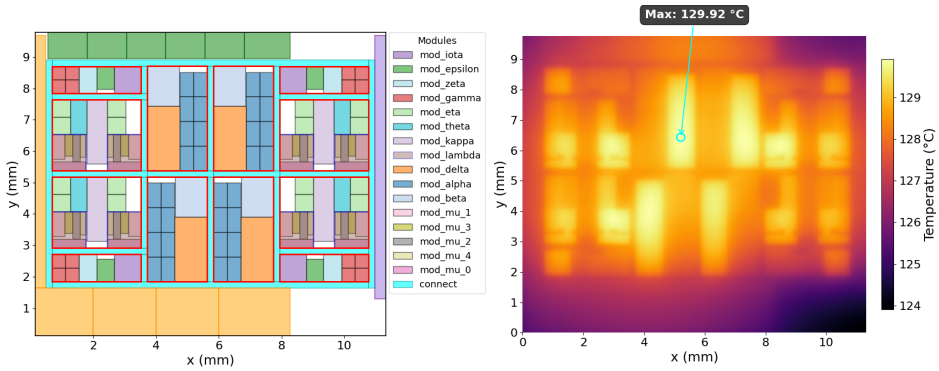


Figure 4.14: 12b zone floorplan (a) placement and (b) thermal map.

Table 4.6: SARL Optimization metrics of best floorplan for different zone divisions and 5 grouping suggestions

Zone Scheme	Area Utilization	HPWL(μm)	ΔT (°C)	Cost	Runtime(s)
1	0.8992	37610	10.97	0.1895	671
4	**0.9512**	57474	11.10	0.1691	415
8	0.9455	38093	10.62	0.1623	545
12a #1	0.9462	**30575**	10.33	**0.1569**	709
12a #2	0.9357	32734	**9.85**	**0.1569**	693
12b	0.9350	40224	9.92	0.1596	599

From the results in table 4.6 it can be seen that the 12a zone implementation gives the best overall result. The 4 zone implementation offers the best area utilization but has significant increase in higher thermal delta and wirelength. Comparing the two results of 12a, the second result of 12a has a lower thermal delta but it comes at the cost of slightly worse area utilization and wirelength.

4.5 Tradeoffs

In the previous section we generated the best results over multiple using a fixed predetermined cost. In this section we instead compare the tradeoffs of varying the cost parameters and the result this has on the floorplan. All results presented in this section are done on the 12a floorplan zone layout. The first results display the best area, wirelength and thermal achievable through setting the weight to 1, respectively 0 for the other parameters.

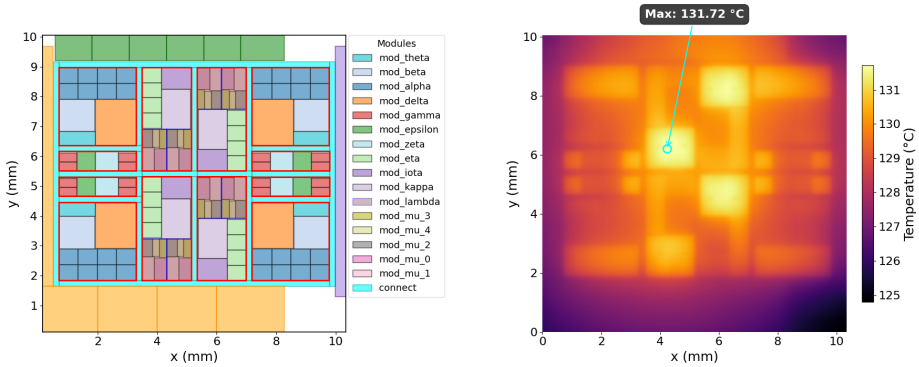


Figure 4.15: Best area floorplan

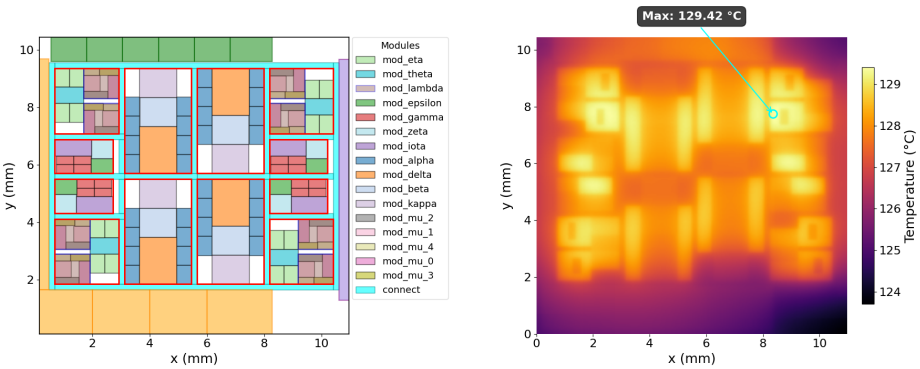


Figure 4.16: Best wirelength floorplan

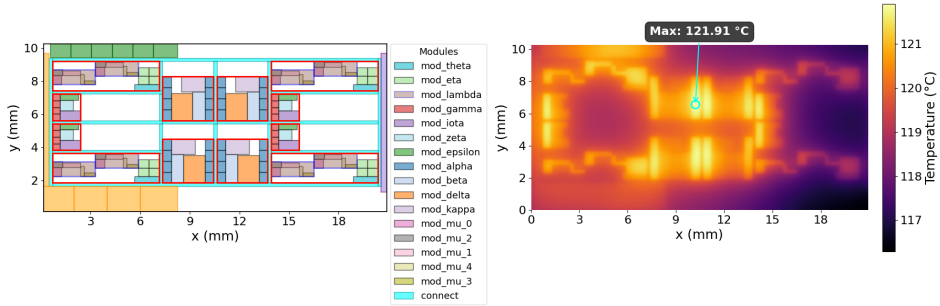


Figure 4.17: Best thermal floorplan

Table 4.7: Pure Weight Comparison Metrics

Pure Weight	Area Utilization	HPWL(μm)	ΔT (°C)	Cost
Area	**0.9875**	410589	11.72	0.23
Wirelength	0.8864	**29269**	9.42	0.18
Thermal	0.5160	244505	**1.91**	0.31

From the results in table 4.7 , we can see that when changing which parameter to optimize for that the algorithm adapts and optimizes based on that requirement. It is also clear visually that Figure 4.15 has the best area, Figure 4.17 has the best thermal and that 4.16 could have the best wirelength. To get a more general understanding of how the weights affect the resulting floorplan, they can be visualized through ternary plots. The data from the ternary plots consists of a weight sweep of all combinations of tenths that add to 1.0 such as $0.2+0.7+0.1 = 1.0$ with each combination being evaluated 12 times at 12 different seeds.

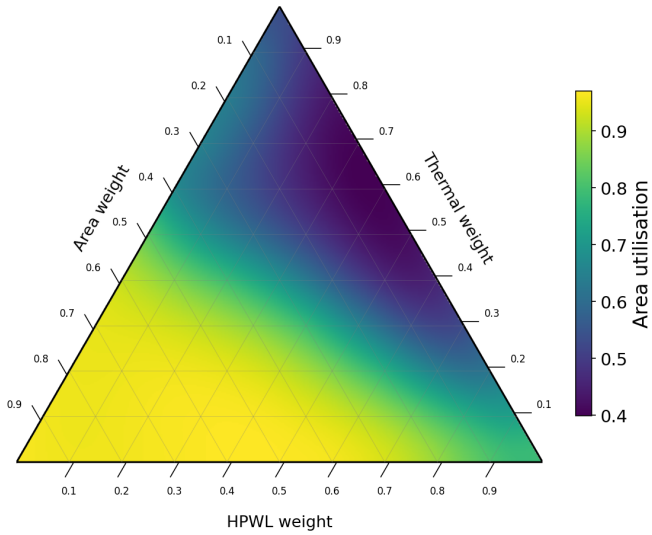


Figure 4.18: Ternary plot of how varying weights affect the area utilization.

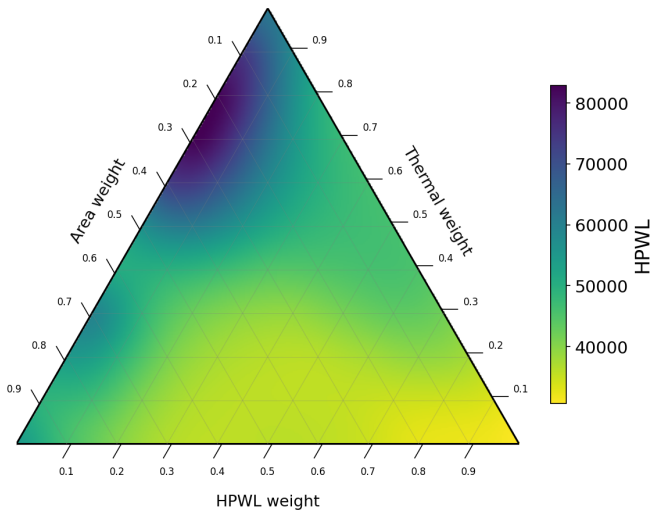


Figure 4.19: Ternary plot of how varying weights affect the wirelength.

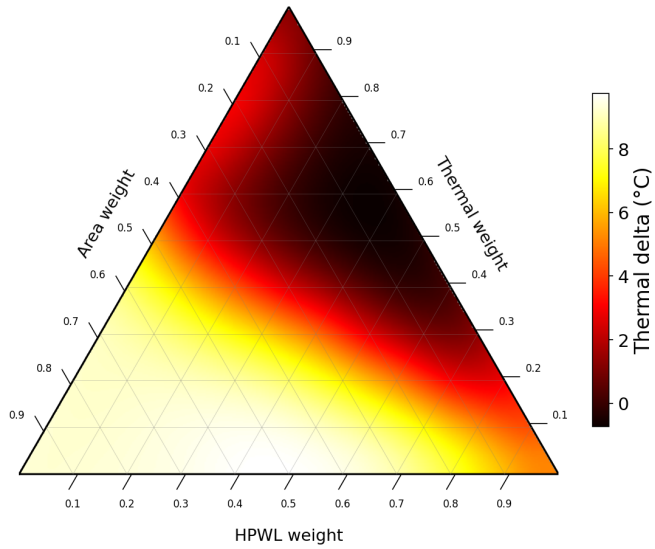


Figure 4.20: Ternary plot of how varying weights affect the temperature.

From Figure 4.18, 4.19 and 4.20 it is clearer how modifying the weights affects the final costs, which gives patterns such as a clear diagonal decision boundary in the tradeoff between temperature and area. To visualize the tradeoffs between the area, wirelength and temperature further, we can plot the Pareto front of the costs. The Pareto front are combinations of weights such that no other combination produces better results in all the costs. The 3D Pareto front is shown in Figure 4.21 and marginal Pareto fronts in 2D are shown in Figure 4.22, 4.23 and 4.24.

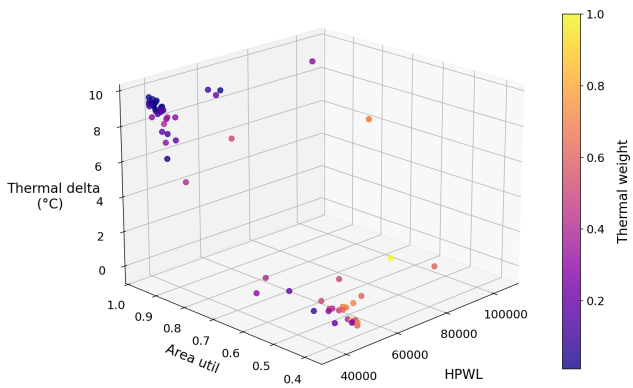


Figure 4.21: The 3D Pareto front such that no data point has better area, thermal and wirelength than any other point.

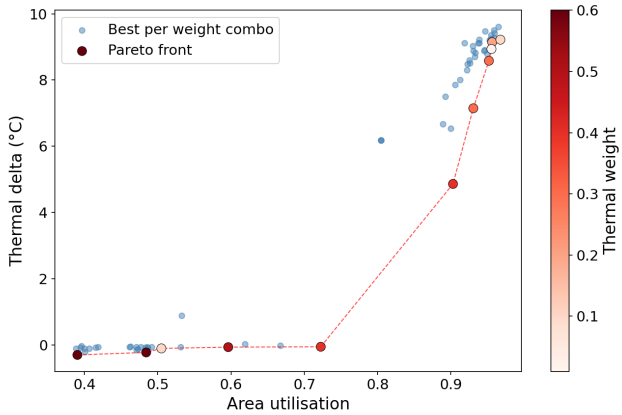


Figure 4.22: The 2D Pareto front between area and temperature such that no data point has better area and temperature than any other point. The blue dots are the best floorplan per weight combination (computed with previous section weights) over 12 runs.

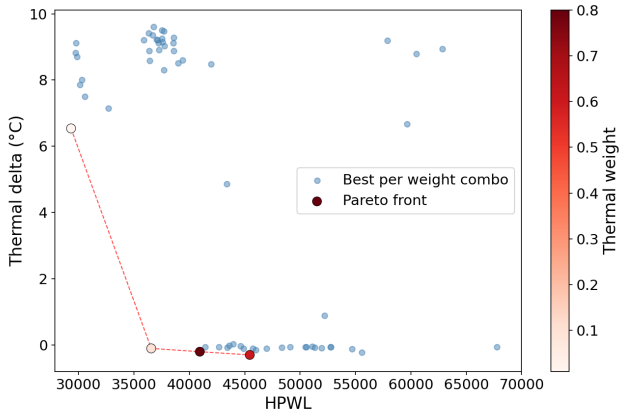


Figure 4.23: The 2D Pareto front between wirelength and temperature such that no data point has better wirelength and temperature than any other point.

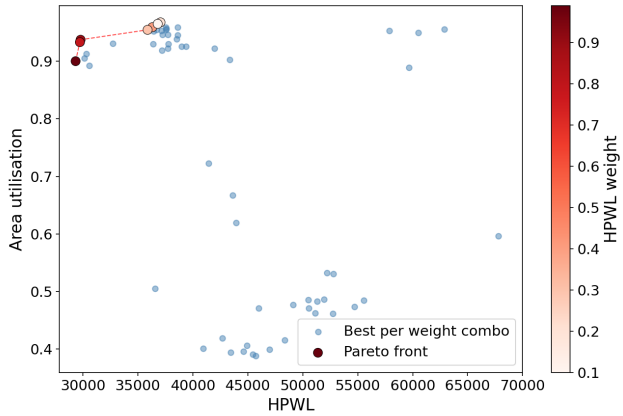


Figure 4.24: The 2D Pareto front between area and wirelength such that no data point has better area and wirelength than any other point.

In Figure 4.21 it can be seen that there are two clear groupings in the Pareto front which is low area utilization, temperature and wirelength, as well as high area, high temperature but still low wirelength. This indicates that the main tradeoff occurs between area and temperature which is also visible in Figure 4.22 which has a well defined separation line. Figure 4.23 shows there is a tradeoff between wirelength and temperature, although quite low wirelengths can be achieved even without significant increase in temperature. Finally, in Figure 4.24 it is clear that wirelength and area are very correlated, since the Pareto boundary is at the very limit of the data indicating that high area utilization often means low wirelength. A detailed table of how weight combinations affect the cost is presented in Appendix A in table A1.

Chapter 5
Discussion and Conclusion

5.1 Discussion

5.1.1 Thermal simulator

In our results we were able to demonstrate how to use our tool to create thermally aware floorplans in a generalizable way. In order to verify our model, we compare our thermal simualtor with HotSpot in Figure 4.1. HotSpot appears to create the same thermal distribution as our model, only it looks a lot more blurred. We believe that this could be from HotSpot modelling at a greater z-level than our simulator causing the temperatures of different blocks to merge more than our algorithm. The fact that our simualtor is more fine-grained is a positive thing in regard to optimization, since it will make it clearer which move and block affects the temperature of the floorplan the most.

We also compare our thermal simulator with HotSpot by running 1000 simulations, and conclude that the mean maximum temperature difference is 0.25 ± 0.97. Considering that the standard deviation of HotSpot itself on the training data is 1.39 this indicates that our model is more accurate than just predicting the mean. This can also be seen in Figure 4.2 where the location of the hottest maximum temperature and hottest maximum interior temperature appear to follow an exponential distribution with a median difference of around 0.6mm which is quite accurate on a 10x10mm die given that the typical block area is around 0.5mm^2.

5.1.2 SARL

One of the main goals of this thesis was to create an algorithm that can automatically create thermal aware floorplans, while being generalizable to work on new topologies. We did so using simulated annealing with reinforcement learning (SARL) and present the exact architecture hyperparameters in table 4.2. One noticeable thing with the hyperparameters is that the neural network isn't that large and contains very few layers which gives even more strength to the results that are presented. We trained the model on a proprietary chip design, but could be generalized to any chip.

In Figure 4.3 the training and validation loss for the weighted negative log loss (WNLL) is shown. The training loss behaves as expected with a weight decay with the loss oscillating and gradually decreasing until a minimum is found. A more exciting result is how the validation loss appears to exactly match the training loss with a single epoch offset. We believe that this is due to the validation data being significantly correlated to the training data despite them using different SA trajectories; the training and validation data were both based on the same floorplan information. This indicates that our model learned all it could from this single ASIC design, and to generalize further

would need more datasets to learn from. For such a small neural network to learn all available information from a single ASIC is impressive, and could point to future work on training on more designs.

Despite our model only being trained on a proprietary design, when evaluated on the proprietary design and the custom dataset it outperforms SA in both cases as seen in Figure 4.4 and 4.6. The SA is outperformed in all 5 group suggestions for both designs which shows that our SARL algorithm can handle various group connections as well as various zone and module configurations. In Figure 4.5 and 4.7 it can be seen that the final result is statistically different for all the thermal suggestions and non-thermal suggestions indicating that it has truly learned to improve over the baseline model. One interesting thing to note is that the difference between the models significantly increases in the first half of the run when the simulated annealing weight is high, whereas it stays constant when the simulated annealing weight is low. This could be due to lower temperatures near convergence gives the optimizer a smaller improvement space that both SA and SARL exploit similarly. Earlier in the run certain moves lead to greater improvements and SARL finds the ones with the largest potential improvements.

Another key insight that is shown from SARL is presented in table 4.5 that shows that SA runs at 188.5 iterations/second whereas SARL runs at 130.0 iterations per second. This is a quite small difference, and shows that replacing a simple simulated annealing run with our generalizable algorithm doesn't add a large time increase. The simulations were run on GPU which helps speed up the neural network inference.

5.1.3 Final results

In section 4.4 we present how our tool can be used to produce viable floorplans from a blank canvas. Initially we run the optimizer with a single zone to get an estimate of group areas and the overall look of the floorplan. This is combined with a connectivity graph and using these two figures (Figure 4.8 and Figure 4.9) it becomes clear that floorplans with even divisions of 4, 8 and 12 are reasonable. Although the zone building logic requires manual work that could be automated in future works, it is a good starting point for any floorplan designer to get an overview of the chip.

After the single zone floorplan along with the Louvain clustering graph is done, we tested the 4, 8, 12a and 12b zone layouts to get the best result. From a thermal perspective, we observe in all our floorplans that the optimizer tries to separate the most power dense blocks aways from each other without increasing the other costs significantly. For our custom dataset, we observe that module alpha and module lambda have the highest power density. From the floorplans we can see that their instances are critical to the peak temperatures and thus kept away from each other whenever possible. Regarding the wirelength, we observe that we get the best results in 12a zone(Figure 4.12). This is a result of the right grouping composition as well as relative placements of groups next to each other. The best result is given by an alternate clustering suggestion which has a worse modularity score than the baseline but minimizes costs more depending on relative placement with other groups. This shows that the utility of trying out alternate grouping suggestions, and their efficacy in generating lower cost floorplans despite their slightly worse modularity scores. By generating different floorplans with different layouts it is easier for a designer to determine what would work the best.

In table 4.7 we compare optimizing purely for area, wirelength and thermal. The table shows that by modifying the weights we get floorplans that optimize for different costs which can also be seen visually in Figure 4.15, 4.17 and 4.16. A more detailed description for this is presented in the ternary plots in Figure 4.18, 4.19 and 4.20 and most interestingly is that the boundary for area and thermal heavily coincide. When the

area utilization improves in one direction the temperature worsens and vice-versa. This is also clear in Figure 4.22 where the Pareto demonstrates the tradeoffs between area and temperature. Furthermore, the floorplans in Figure 4.12 and 4.13 provide an example of two floorplans with the exact same total cost but have different area, wirelength and tempearture due to the tradeoffs of the Pareto front in Figure 4.21 All in all it is clear that modifying the weights in the cost function produces different floorplans with varying properties, and that there are inherent tradeoffs within these floorplans.

5.2 Conclusion

In conclusion we have designed a method to produce thermal aware floorplans that can be generalized to new ASICs. Our algorithm works as intended with the generated floorplans being usable in real designs due to their hierachical structures and low costs. We showed that significant temperature reductions can be achieved if temperature is included as a constraint rather than only optimizing for area and wirelength . We have also showed that there is a trade-off that is dependent on what you optimize for. Our thermal simulator that depends on analytical solutions rather than a lookup table is also unique and very fast, and it can inherently enforce the boundary conditions as well rather than just approximating them. Comparing our thermal simulator with Hotspot, we have also seen it has significant accuracy in locating and measuring peak temperatures with low mean errors. Finally, we presented SARL and showed that this method outperforms simulated annealing in accuracy and it can be generalized without a large decrease in speed. All in all, the method and results align with the goals of what was set out to be done.

5.2.1 Future works

Regarding the future of our work, we believe that our methods can be used to design thermally aware floorplans that are usable in practice. There are many future directions that can be taken to improve upon our work: one would be to make the zone divisions more automatic rather than a user determining the zones. Another improvement is a more accurate wirelength calculation algorithm can also determine approximate routing paths through connects. A more accurate routing algorithm could also take into account feedthrough wires, and how to minimize wires going through other blocks. An improvement for accuracy in the thermal simulator would be to make the thermal conductivity constant temperature dependent, but this would result in increased computational cost and the need for running multiple iterations. The optimizer is also very crude and only trained on a single floorplan design, but with more compute and more floorplans it could learn to generalize even better. Additionally, the refinement step could be improved to handle joint optimization, and the the clustering logic could be expanded to cluster not only based on wirelength but also on power density.

The methods presented in this thesis lay a foundation for thermally aware ASIC design, and as chip complexity continues to grow, approaches like ours will become not just useful, but necessary.

Appendix A

Table A1: Results by weight configuration

w_{area}	w_{hpwl}	w_{thermal}	Area	HPWL	Thermal
0.0	0.0	1.0	0.47 ± 0.04	337226 ± 260848	2.44 ± 0.78
0.0	0.1	0.9	0.37 ± 0.02	85239 ± 16300	-0.04 ± 0.04
0.0	0.2	0.8	0.37 ± 0.01	87683 ± 20077	-0.05 ± 0.07
0.0	0.3	0.7	0.37 ± 0.01	85373 ± 17520	-0.02 ± 0.03
0.0	0.4	0.6	0.37 ± 0.01	84784 ± 17658	-0.05 ± 0.08
0.0	0.5	0.5	0.37 ± 0.02	84957 ± 17671	-0.01 ± 0.02
0.0	0.6	0.4	0.37 ± 0.02	83387 ± 16844	-0.03 ± 0.03
0.0	0.7	0.3	0.37 ± 0.01	80064 ± 16824	-0.01 ± 0.04
0.0	0.8	0.2	0.37 ± 0.02	81125 ± 17787	-0.00 ± 0.04
0.0	0.9	0.1	0.42 ± 0.05	72127 ± 12624	1.04 ± 0.98
0.0	1.0	0.0	0.81 ± 0.06	55858 ± 9397	8.33 ± 0.95
0.1	0.0	0.9	0.74 ± 0.02	310705 ± 161757	6.48 ± 0.39
0.1	0.1	0.8	0.41 ± 0.03	94012 ± 19658	-0.03 ± 0.04
0.1	0.2	0.7	0.40 ± 0.03	87679 ± 16845	-0.01 ± 0.03
0.1	0.3	0.6	0.40 ± 0.03	90892 ± 17304	-0.02 ± 0.03
0.1	0.4	0.5	0.40 ± 0.04	87883 ± 17094	-0.02 ± 0.05
0.1	0.5	0.4	0.39 ± 0.01	86433 ± 18370	-0.00 ± 0.04
0.1	0.6	0.3	0.42 ± 0.04	84320 ± 17510	0.02 ± 0.07
0.1	0.7	0.2	0.49 ± 0.05	78057 ± 13700	0.94 ± 1.14
0.1	0.8	0.1	0.86 ± 0.03	59511 ± 10552	8.74 ± 0.55
0.1	0.9	0.0	0.91 ± 0.02	58671 ± 10517	9.79 ± 0.36
0.2	0.0	0.8	0.74 ± 0.02	303572 ± 157723	6.48 ± 0.39
0.2	0.1	0.7	0.43 ± 0.04	114038 ± 39933	-0.01 ± 0.04
0.2	0.2	0.6	0.44 ± 0.04	94325 ± 18350	-0.01 ± 0.03
0.2	0.3	0.5	0.44 ± 0.04	89036 ± 19383	-0.02 ± 0.03
0.2	0.4	0.4	0.45 ± 0.04	89400 ± 17518	0.00 ± 0.05
0.2	0.5	0.3	0.54 ± 0.07	81113 ± 15437	1.59 ± 1.58
0.2	0.6	0.2	0.86 ± 0.02	60792 ± 10286	8.58 ± 0.54
0.2	0.7	0.1	0.90 ± 0.02	58715 ± 10193	9.55 ± 0.33
0.2	0.8	0.0	0.90 ± 0.02	59066 ± 9734	9.62 ± 0.31
0.3	0.0	0.7	0.74 ± 0.02	303572 ± 157723	6.48 ± 0.39

Continued on next page

Table A1 – continued from previous page

w_{area}	w_{hpwl}	w_{thermal}	Area	HPWL	Thermal
0.3	0.1	0.6	0.47 ± 0.02	121024 ± 46179	-0.03 ± 0.06
0.3	0.2	0.5	0.48 ± 0.02	98895 ± 18749	0.11 ± 0.37
0.3	0.3	0.4	0.55 ± 0.07	82066 ± 13674	1.78 ± 1.63
0.3	0.4	0.3	0.86 ± 0.05	63144 ± 9951	8.55 ± 0.83
0.3	0.5	0.2	0.89 ± 0.02	60857 ± 11006	9.10 ± 0.50
0.3	0.6	0.1	0.90 ± 0.02	60620 ± 11694	9.54 ± 0.24
0.3	0.7	0.0	0.91 ± 0.02	59137 ± 10292	9.82 ± 0.62
0.4	0.0	0.6	0.48 ± 0.01	292589 ± 159009	0.00 ± 0.03
0.4	0.1	0.5	0.50 ± 0.04	103907 ± 26320	0.38 ± 0.89
0.4	0.2	0.4	0.85 ± 0.06	66305 ± 9602	8.27 ± 1.23
0.4	0.3	0.3	0.89 ± 0.02	64114 ± 10174	9.19 ± 0.38
0.4	0.4	0.2	0.90 ± 0.02	59665 ± 9730	9.48 ± 0.29
0.4	0.5	0.1	0.91 ± 0.02	61108 ± 10820	9.66 ± 0.47
0.4	0.6	0.0	0.92 ± 0.03	63967 ± 19433	10.12 ± 0.74
0.5	0.0	0.5	0.86 ± 0.03	126536 ± 65596	8.32 ± 0.61
0.5	0.1	0.4	0.90 ± 0.02	77635 ± 36619	9.17 ± 0.45
0.5	0.2	0.3	0.89 ± 0.02	67222 ± 13813	9.21 ± 0.50
0.5	0.3	0.2	0.90 ± 0.02	62046 ± 10846	9.65 ± 0.64
0.5	0.4	0.1	0.92 ± 0.02	65474 ± 20411	10.03 ± 0.69
0.5	0.5	0.0	0.92 ± 0.02	64894 ± 18892	10.27 ± 0.80
0.6	0.0	0.4	0.91 ± 0.01	137689 ± 89075	9.44 ± 0.40
0.6	0.1	0.3	0.90 ± 0.02	92671 ± 49075	9.54 ± 0.70
0.6	0.2	0.2	0.91 ± 0.02	69279 ± 26131	9.65 ± 0.61
0.6	0.3	0.1	0.92 ± 0.02	71672 ± 32317	10.05 ± 0.64
0.6	0.4	0.0	0.92 ± 0.02	80610 ± 40575	10.33 ± 0.74
0.7	0.0	0.3	0.92 ± 0.02	167573 ± 31574	9.77 ± 0.52
0.7	0.1	0.2	0.92 ± 0.02	101396 ± 49138	9.76 ± 0.69
0.7	0.2	0.1	0.93 ± 0.03	89436 ± 49613	10.42 ± 0.87
0.7	0.3	0.0	0.93 ± 0.02	84789 ± 40707	10.34 ± 0.67
0.8	0.0	0.2	0.92 ± 0.02	168502 ± 93707	10.02 ± 0.76
0.8	0.1	0.1	0.93 ± 0.02	103396 ± 43776	10.26 ± 0.72
0.8	0.2	0.0	0.93 ± 0.02	91368 ± 44099	10.53 ± 0.72
0.9	0.0	0.1	0.93 ± 0.02	116872 ± 62534	10.12 ± 0.74
0.9	0.1	0.0	0.93 ± 0.02	186055 ± 119134	10.19 ± 0.68

Bibliography

[1] Hui-Fang Tsao, Pang-Yen Chou, Shih-Lun Huang, Yao-Wen Chang, Mark Po-Hung Lin, Duan-Ping Chen, and Dick Liu. A corner stitching compliant b-tree representation and its applications to analog placement. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 507–511, 2011.

[2] Yun-Chih Chang, Yao-Wen Chang, Guang-Ming Wu, and Shu-Wei Wu. B*-trees: a new representation for non-slicing floorplans. In *Proceedings 37th Design Automation Conference*, pages 458–463, 2000.

[3] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nova, and et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, Jun 2021.

[4] Jianye Hao Ruizhe Zhong Hongyao Tang Zhentao Tang Mingxuan Yuan Junchi Yan Pengyi Li, Shixiong Kai. Core: Collaborative optimization with reinforcment learning and evolutionary algorithm for floorplanning. Not published but presented at Neurips 2025, 2025.

[5] Shenglu Yu, Shimin Du, and Chang Yang. A deep reinforcement learning floorplanning algorithm based on sequence pairs. *Applied Sciences*, 14(7), 2024.

[6] Yuanyuan Duan, Xingchen Liu, Zhiping Yu, Hanming Wu, Leilai Shao, and Xiaolei Zhu. Rlplanner: Reinforcement learning based floorplanning for chiplets with fast thermal analysis. In *2024 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–2, 2024.

[7] R.H.J.M. Otten. Automatic floorplan design. In *19th Design Automation Conference*, pages 261–267, 1982.

[8] Lian He. PhD thesis, Memorial University of Newfoundland, 2021.

[9] Chung Kuan-Cheng. Floorplan representation in vlsi, 2009.

[10] D.F. Wong and C.L. Liu. A new algorithm for floorplan design. In *23rd ACM/IEEE Design Automation Conference*, pages 101–107, 1986.

[11] X. Hong, Sheqin Dong, Gang Huang, Y. Cai, Chung-Kuan Cheng, and Jun Gu. Corner block list representation and its application to floorplan optimization. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 51(5):228–233, 2004.

[12] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. Rectangle-packing-based module placement. In *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pages 472–479, 1995.

[13] Xiaoping Tang and D.F. Wong. Fast-sp: a fast algorithm for block placement based on sequence pair. In *Proceedings of the ASP-DAC 2001. Asia and South Pacific Design Automation Conference 2001 (Cat. No.01EX455)*, pages 521–526, 2001.

[14] S. Nakatake, K. Fujiyoshi, H. Murata, and Y. Kajitani. Module placement on bsg-structure and ic layout applications. In *Proceedings of International Conference on Computer Aided Design*, pages 484–491, 1996.

[15] Pei-Ning Guo, Chung-Kuan Cheng, and T. Yoshimura. An o-tree representation of non-slicing floorplan and its applications. In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pages 268–273, 1999.

[16] Jai-Ming Lin and Yao-Wen Chang. Tcg: A transitive closure graph-based representation for general floorplans. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(2):288–292, 2005.

[17] Jai-Ming Lin and Yao-Wen Chang. Tcg-s: orthogonal coupling of p/sup */-admissible representations for general floorplans. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(6):968–980, 2004.

[18] F. Balasa and K. Lampaert. Symmetry within the sequence-pair representation in the context of placement for analog design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(7):721–731, 2000.

[19] Shinichi Koda, Chikaaki Kodama, and Kunihiro Fujiyoshi. Linear programming-based cell placement with symmetry constraints for analog ic layout. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(4):659–668, 2007.

[20] Yingxin Pang, Florin Balasa, Koen Lampaert, and Chung-Kuan Cheng. Block placement with symmetry constraints based on the o-tree non-slicing representation. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, page 464–467, New York, NY, USA, 2000. Association for Computing Machinery.

[21] F. Balasa, S.C. Maruvada, and K. Krishnamoorthy. On the exploration of the solution space in analog placement with symmetry constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(2):177–191, 2004.

[22] Jai-Ming Lin, Guang-Ming Wu, Yao-Wen Chang, and Jen-Hui Chuang. Placement with symmetry constraints for analog layout design using tcg-s. In *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.*, volume 2, pages 1135–1138 Vol. 2, 2005.

[23] Po-Hung Lin, Yao-Wen Chang, and Shyh-Chang Lin. Analog placement based on symmetry-island formulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(6):791–804, 2009.

[24] Linfu Xiao and Evangeline F.Y. Young. Analog placement with common centroid and 1-d symmetry constraints. In *2009 Asia and South Pacific Design Automation Conference*, pages 353–360, 2009.

[25] Qiang Ma, Linfu Xiao, Yiu-Cheong Tam, and Evangeline F. Y. Young. Simultaneous handling of symmetry, common centroid, and general placement constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(1):85–95, 2011.

[26] I-Peng Wu, Hung-Chih Ou, and Yao-Wen Chang. Qb-trees: Towards an optimal topological representation and its applications to analog layout designs. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[27] E.F.Y. Young, C.C.N. Chu, and M.L. Ho. Placement constraints in floorplan design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(7):735–745, 2004.

[28] Simulated-annealing channel router. In *Unknown Host Publication Title*, pages 226–228. IEEE, 1985.

[29] S Adya, HH Chan, and I Markov. Parquet: Fixed-outline foorplanner. online: http://vlsicad. eecs. umich. edu. *BK/parquet*.

[30] Naushad Manzoor Laskar, Rahul Sen, P.K. Paul, and K.L. Baishnab. A survey on vlsi floorplanning: Its representation and modern approaches of optimization. In *2015 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*, pages 1–9, 2015.

[31] M. Rebaudengo and M.S. Reorda. Gallo: a genetic algorithm for floorplan area optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(8):943–951, 1996.

[32] S. Nakaya, T. Koide, and S. Wakabayashi. An adaptive genetic algorithm for vlsi floorplanning based on sequence-pair. In *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 3, pages 65–68 vol.3, 2000.

[33] Tsung-Ying Sun, Sheng-Ta Hsieh, Hsiang-Min Wang, and Cheng-Wei Lin. Floorplanning based on particle swarm optimization. In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, pages 5 pp.–, 2006.

[34] Rajendra Bahadur Singh. A binary particle swarm optimization for ic floorplanning. *Recent Advances in Computer Science and Communications*, 13(1):13–21, 2020.

[35] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[36] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, and et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016.

[37] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *nature*, 575(7782):350–354, 2019.

[38] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.

[39] Yao Lai, Yao Mu, and Ping Luo. Maskplace: Fast chip placement via reinforced visual representation learning. *ArXiv*, abs/2211.13382, 2022.

[40] Yao Lai, Jinxin Liu, Zhentao Tang, Bin Wang, Jianye Hao, and Ping Luo. Chipformer: transferable chip placement via offline decision transformer. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[41] Anthony Agnesina, Puranjay Rajvanshi, Tian Yang, Geraldo Pradipta, Austin Jiao, Ben Keller, Brucek Khailany, and Haoxing Ren. Autodmp: Automated dreamplace-based macro placement. ISPD '23, page 149–157, New York, NY, USA, 2023. Association for Computing Machinery.

[42] Zijie Geng, Jie Wang, Ziyan Liu, Siyuan Xu, Zhentao Tang, Mingxuan Yuan, Jianye Hao, Yongdong Zhang, and Feng Wu. Reinforcement learning within tree search for fast macro placement. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

[43] Zhuolun He, Yuzhe Ma, Lu Zhang, Peiyu Liao, Ngai Wong, Bei Yu, and {Martin D. F.} Wong. Learn to floorplan through acquisition of effective local search heuristics. In *Proceedings of The IEEE 38th International Conference on Computer Design, ICCD 2020*, Proceedings - IEEE International Conference on Computer Design (ICCD): VLSI in Computers and Processors, pages 324–331, United States, October 2020. IEEE. Funding Information: This work is partially supported by The Research Grants Council of Hong Kong SAR (No. CUHK14209420). Publisher Copyright: © 2020 IEEE.; 38th IEEE International Conference on Computer Design, ICCD 2020 ; Conference date: 18-10-2020 Through 21-10-2020.

[44] Qi Xu, Hao Geng, Song Chen, Bo Yuan, Cheng Zhuo, Yi Kang, and Xiaoqing Wen. Goodfloorplan: Graph convolutional network and reinforcement learning-based floorplanning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(10):3492–3502, 2022.

[45] Mohammad Amini, Zhanguang Zhang, Surya Penmetsa, Yingxue Zhang, Jianye Hao, and Wulong Liu. Generalizable floorplanner through corner block list representation and hypergraph embedding. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '22, page 2692–2702, New York, NY, USA, 2022. Association for Computing Machinery.

[46] Song Bai, Feihu Zhang, and Philip H.S. Torr. Hypergraph convolution and hypergraph attention. *Pattern Recognition*, 110:107637, 2021.

[47] Brian W. Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[48] Charles M. Fiduccia and Robert M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference (DAC)*, pages 175–181, 1982.

[49] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Application in VLSI domain. In *Proceedings of the 34th Design Automation Conference (DAC)*, pages 526–529, 1997.

[50] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, Oct 2008.

[51] Thomas Aynaud. python-louvain x.y: Louvain algorithm for community detection. `https://github.com/taynaud/python-louvain`, 2020.

[52] Andrzej Kos. An approach to thermal placement in power electronics using neural networks. *1993 IEEE International Symposium on Circuits and Systems*, pages 2427–2430 vol.4, 1993.

[53] KAI-YUAN CHAO and D. F. WONG. Floorplan design with low power considerations. *International Journal of High Speed Electronics and Systems*, 07(02):305–322, Jun 1996.

[54] {Chris C.N.} Chu and {D. F.} Wong. A matrix synthesis approach to thermal placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(11):1166–1174, 1998. Manuscript received April 24, 1998. This work was supported in part by the Texas Advanced Research Program under Grant 003658288. This paper was recommended by Associate Editor C. K. Cheng. The authors are with the Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712 USA. Publisher Item Identifier S 0278-0070(98)08594-7.

[55] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Temperature-aware microarchitecture. *SIGARCH Comput. Archit. News*, 31(2):2–13, May 2003.

[56] Karthik Sankaranarayanan, Sivakumar Velusamy, Mircea Stan, and Kevin Skadron. A case for thermal-aware floorplanning at the microarchitectural level. *The Journal of Instruction-Level Parallelism*, 7, Oct 2005.

[57] Yongkui Han and Israel Koren. Simulated annealing based temperature aware floorplanning. *J. Low Power Electronics*, 3:141–155, 08 2007.

[58] Xiaodao Chen, Lizhe Wang, Albert Y. Zomaya, Lin Liu, and Shiyan Hu. Cloud computing for vlsi floorplanning considering peak temperature reduction. *IEEE Transactions on Emerging Topics in Computing*, 3(4):534–543, 2015.

[59] Xinfei Liu, Siting Liu, Bei Yu, Song Chen, and Qi Xu. Theplace: Thermal-aware placement with operator learning-based ultra-fast simulator. *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, 2025.

[60] Yikai Liu, Jindong Zhou, Jiayi Li, and Pingqiang Zhou. Fast thermal-driven 3d fixed-outline floorplanning by learning-based thermal analysis. In *2025 IEEE 16th International Conference on ASIC (ASICON)*, pages 1–4, 2025.

[61] Daniel Almeida Fagundes. Electrostatic potential of a rectangular uniformly charged plate: exact solution and limiting cases. *European Journal of Physics*, 43(1):015203, nov 2021.

[62] Gunnar Sparr and Annika Sparr. *Kontinuerliga System*. Studentlitteratur, 2000.

[63] GSRC Bookshelf Benchmark Suite. Gsrc bookshelf floorplanning benchmarks (historical). Benchmarks commonly referred to as the GSRC/Bookshelf set. Access via institutional mirror, Open-source mirrors, or the Internet Archive.

[64] MCNC Benchmark Suite. Mcnc placement/floorplanning benchmarks (historical). Benchmark suite widely cited in placement/floorplanning literature; original archives may be mirrored. Access via institutional mirror or archive.