University of Thessaly

A Thesis presented for the degree of
Diploma of Science in Computer and Communications Engineering

By

XANTHOS VLACHOS

Department of Electrical and Computer Engineering
UNIVERSITY OF THESSALY

Supervisors

Christos Sotiriou, Associate Professor
Nestoras Eumorfopoulos, Assistant Professor

OCTOBER 2016

Word count: eleven thousand and eighty

"Implementation and Analysis of Placement Algorithms based on methods of Mechanics (Force Directed) for Microelectronic Circuits"

’Υλοποίηση και Ανάλυση Αλγορίθμων Τοποθέτησης βασισμένοι σε μεθόδους Μηχανικής **(Force Directed)** για Μικροηλεκτρονικά Κυκλώματα’

Abstract

Nowadays, Electronic Design Automation (EDA) is required for every IC to be designed and fabricated. Placement is a paramount stage in the manufacturing process. One of the Placement objectives is to target the best and Minimum Total Wirelength (TWL). Both Analytical and Combinatorial methods do exist for Placement process. Although Combinatorial techniques, which are characterized of low complexity and slow execution, give an exact solution, Analytical ones, and especially Linear ones, are much faster by giving a very good approximation of the solution.

Being challenged by a Force-directed Analytical Placer, with purely Linear complexity, called *Kraftwerk2*, we propose a new extended version *Kraftwerk2++* that supports Sparse Linear solving, faster Convergence using a Maximum Density metric and both Poisson and Gaussian Blur formulation that the algorithm's Forces derive from. Furthermore, an enriched Quality Control system is analyzed and proposed that aims in a correct, concrete and of high quality placement. An attempt to support Clusters is presented, too. Last but not least, it should be mentioned that Kraftwerk2 has three degrees of Freedom that need to be balanced, which are not in the original algorithm. Hence, we inserted Combinatorial Logic in our iterative Kraftwerk2++ placer, in order to reject incorrect and invalid local solutions, balancing every placement's solution. Finally, we present results proving the strength of our new KW2++ algorithm.

Περίληψη

Στη σημερινή εποχή, ο Αυτοματισμός στον Ηλεκτρονικό Σχεδιασμό **(EDA)** απαιτείται ώστε να σχεδιαστεί και να κατασκευαστεί ένα ολοκληρωμένο Κύκλωμα **(IC)**. Το στάδιο της Τοποθέτησης των στοιχείων μέσα στον πυρήνα ενός **IC** είναι υψίστης σημασίας. Ένας από τους στόχους της Τοποθέτησης είναι η επίτευξη του καλύτερου και ελάχιστου συνολικού μήκους καλωδίων **(Total Wirelength - TWL)**. Τόσο Αναλυτικές όσο και Συνδυαστικές μέθοδοι υπάρχουν για την διαδικασία της Τοποθέτηση. Παρόλου που οι Συνδυαστικές Τεχνικές, οι οποίες χαρακτηρίζονται από χαμηλή πολυπλοκότητα και αργή εκτέλεση, δίνουν μία ακριβή λύση, οι Αναλυτικές μέθοδοι, και κυρίως οι Γραμμικές, είναι πολύ πιο γρήγορες, δίνοντας μια πολύ καλή προσέγγιση της λύσης.

Επηρεαζόμενοι από έναν Αναλυτικό αλγόριθμο Τοποθέτης, βασισμένο σε μεθόδους Μηχανικής, τον ***Kraftwerk2***, προτείνουμε μια νέα εκτεταμένη έκδοση με όνομα ***Kraftwerk2++***, η οποία υποστηρίζει επίλυση **Sparse** Γραμμικών Προβλημάτων, ταχύτερη σύγκλιση χρησιμοποιώντας μια μετρική **Maximum Density** και έναν φορμαλισμό τόσο της μεθόδου **Poisson** όσο και της **Gaussian Blur** από την οποία προέρχονται οι Δυνάμεις του αλγορίθμου. Επιπρόσθετα, προτείνεται ένα πιο εμπλουτισμένο σύστημα ποιοτικού ελέγχου το οποίο στοχεύει σε μια σωστή, έγκυρη και ποιοτική λύση. Επίσης, παρουσιάζεται μια προσπάθεια για την υποστήριξη **Clusters**, ομαδοποιημένων στοιχείων. Τελικώς είναι πολύ σημαντικό να αναφερθεί ότι ο αλγόριθμος **Kraftwerk2** έχει τρεις βαθμούς ελευθερίας που πρέπει να βρίσκονται σε συνεχή ισορροπία σε όλη την εκτέλεση, πράγμα που δεν εμφανίζεται στην αρχική προσσέγιση του αλγόριθμου. Για το λόγο αυτό, στον δικός μας **Kraftwerk2++** έχει εισαχθεί Συνδυαστική Λογική, ώστε σε όλη την επαναληπτική εκτέλεση του αλγορίθμου να απορρίπτονται λανθασμένες και μη έγκυρες τοπικές λύσεις, εξισορροπώντας έτσι την τελική λύση. Στο τελευταίο κεφάλαιο της διπλωματικής αυτής εργασίας, παρουσιάζουμε αποτελέσματα που αποδεικνύουν τη 'δύναμη' και την αποτελεσματικότητα του νέου μας αλγορίθμου **KW2++**.

Dedication and acknowledgements

I would like to thank my supervisors **Dr. Christos Sotiriou** and **Dr. Nestoras Evmorfopoulos** for their great collaboration, the dozens of hours they spent with me on interesting discussions, their inspiring ideas and the paramount guidance that they provided, which urge me to try harder for the completion of my goals.

I cannot thank enough my collages **Michalis Giaourtas**, **Nikolaos Sketopoulos**, **Stavros Ntentos** and **Angelina Delacura** for their support and insight during the whole period of my master thesis.

Words cannot describe how thankful I am for **my parents** for their support and belief in me during my whole life.

Author's declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work and research. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: ..... DATE: .....

Table of Contents

List of Tables

Table **Page**

List of Figures

Figure **Page**

CHAPTER 1

INTRODUCTION TO EDA

EDA stands for Electonic Design Automation [3, 6]. EDA is becoming ever more important with the continuous scaling of semiconductor devices and the growing complexities of their use in circuits and systems. Demands for lower-power, higher-reliability and more agile electronic systems raise new challenges to both design and design automation of such systems. But, let's get straight to the point by analyzing each word's meaning separately.

Electronic refers to anything electronic, from computer chips, cellular phones, pacemakers, controls for automobiles and satellites to the servers, routers and switches that run the Internet. Everything made by the nearly $1 trillion electronics industry results from designers using EDA tools and services. As electronics become even more complex and pervasive, the EDA industry is more vital to the continued success of the global economy.

Design is the part of the production cycle where creativity, new ideas, ingenuity and inspiration come to the fore. This is also where designers try to model the behavior of their designs and analyze the complex interactions of millions of constituent parts in their designs to ensure completeness, correctness and manufacturability of the final product. Why? Because it is impossibly difficult, expensive and time consuming to "build it first and fix it later." Because the designers in our industry are mostly electrical engineers ("hardware engineers") and computer scientists ("software engineers"), some segments of the EDA industry are also called, "Computer Aided Engineering" (CAE). EDA is also referred to as "Electronic Computer-Aided Design" (ECAD), acknowledging the crucial role EDA plays in the design phase.

Automation! The dramatic increase in complexity - enabled by the relentless onslaught of Moore's Law 1.1 - that designers must tackle in electronics today, drives the need for automation. Engineers need to validate their concepts, model and analyze their designs, identify and eliminate problems before making production commitments. EDA

helps them get it done right.

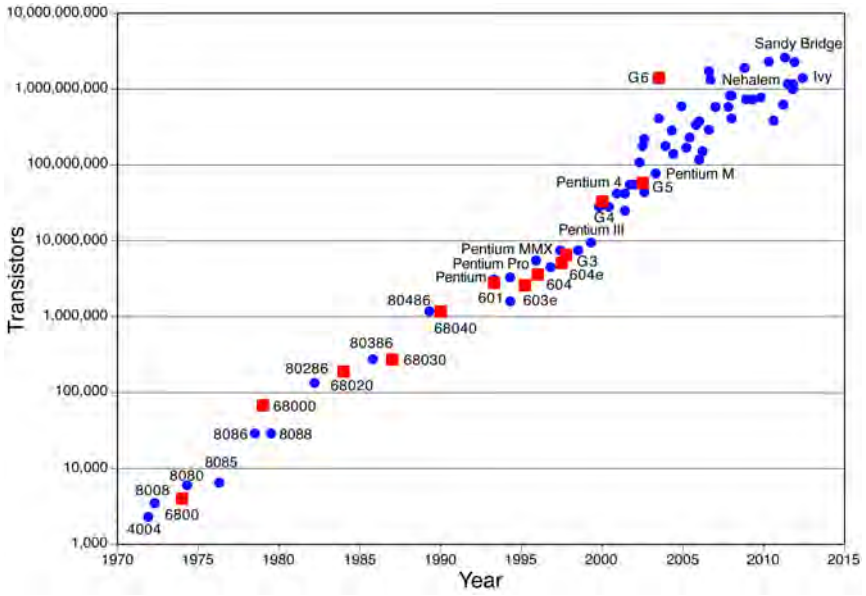


Figure 1.1: Moore's Law

Moore's Law is a trend observed by Intel co-founder Gordon Moore in 1965 in which the number of transistors in integrated circuits doubles every 18 months. For more than 30 years this has been the driving force behind the electronics revolution.

To sum up, EDA is a category of software tools for designing electronic integrated circuits. EDA ease the manufacturing process in all stages and especially in Physical Design, which is my thesis reference area.

1.1 Physical Design - Placement

In integrated circuit design, physical design is a step in the standard design cycle which follows after the circuit design, as analyzed in detail in [3, 6]. At this step, circuit representations of the components (devices and interconnects) of the design are converted into geometric representations of shapes which, when manufactured in the corresponding layers of materials, will ensure the required functioning of the components. This geometric representation is called integrated circuit layout. This step is usually split into several sub-steps, which include both design and verification and validation of the layout, as presented in Figure 1.2.

Modern day Integrated Circuit (IC) design is split up into Front-end design using HDLs, Verification, and Back-end Design or Physical Design [8]. The next step after Physical Design is the Manufacturing process or Fabrication Process that is done in the Wafer Fabrication Houses. Fab-houses fabricate designs onto silicon dies which are then packaged into ICs.

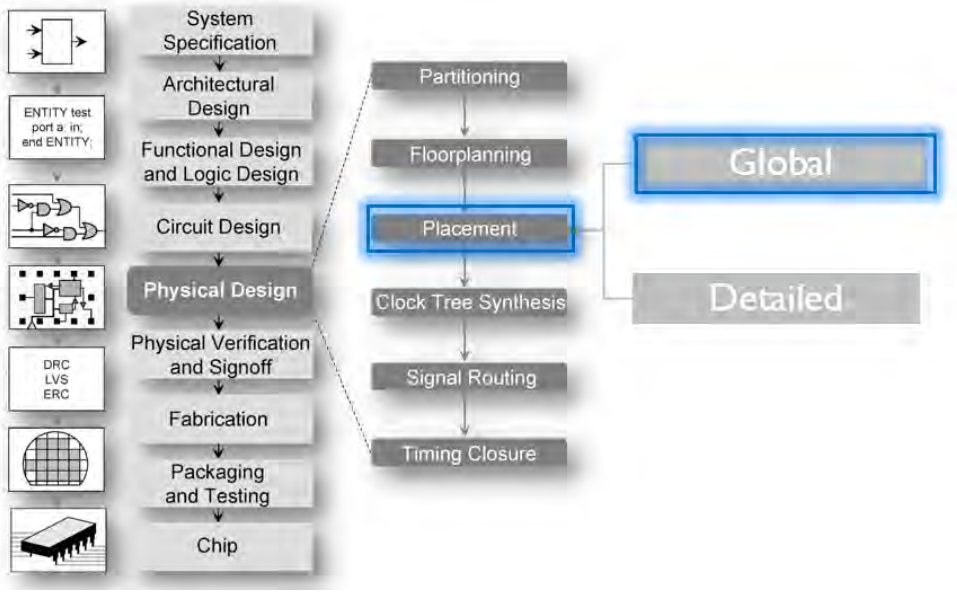


Figure 1.2: Design Flow

Each of the phases mentioned above has Design Flows associated with them. These Design Flows lay down the process and guide-lines/framework for that phase. Physical Design flow uses the technology libraries that are provided by the fabrication houses. These technology files provide information regarding the type of Silicon wafer used, the standard-cells used, the layout rules (like DRC in VLSI), etc.

Becoming more specific, Placement is an essential step in electronic design automation - the portion of the physical design flow that assigns exact locations for various circuit components within the chip's core area. An inferior placement assignment will not only affect the chip's performance but might also make it non-manufacturable by producing excessive wirelength, which is beyond available routing resources. Consequently, a placer must perform the assignment while optimizing a number of objectives to ensure that a circuit meets its performance demands.

Typical placement objectives include:

- Total wirelength: Minimizing the total wirelength, or the sum of the length of all the wires in the design, is the primary objective of most existing placers. This not only helps minimize chip size, and hence cost, but also minimizes power and delay, which are proportional to the wirelength (This assumes long wires have additional buffering inserted; all modern design flows do this.)

- Timing: The clock cycle of a chip is determined by the delay of its longest path, usually referred to as the critical path. Given a performance specification, a placer must ensure that no path exists with delay exceeding the maximum specified delay.

- Congestion: While it is necessary to minimize the total wirelength to meet the total routing resources, it is also necessary to meet the routing resources within various

local regions of the chip's core area. A congested region might lead to excessive routing detours, or make it impossible to complete all routes.

- Power: Power minimization typically involves distributing the locations of cell components so as to reduce the overall power consumption, alleviate hot spots, and smooth temperature gradients.

A secondary objective is placement run-time minimization.

Currently, placement is usually separated into global and detailed placement.

State of the art global placement algorithms include analytic techniques, which approximate the wirelength objective using quadratic or nonlinear formulations, and min-cut placers which use graph partitioning algorithms. The goal of global placement is to find well spread, ideally with no overlaps, placement for the given net-list that attains required objectives such as wirelength minimization or timing specifications. Due to the enormous number of components, these standard cells are places into groups such that the number of connections between groups is minimized. This is solved through circuit partitioning. i.e. A modern placement framework called Kraftwerk combines analytic techniques with fast computational geometry.

Detailed placement uses various kinds of local optimizations, including simulated annealing. Simulated annealing has also been used for the complete placement flow since its proposal as a general combinatorial optimisation technique, before being replaced by analytical and min-cut placers.

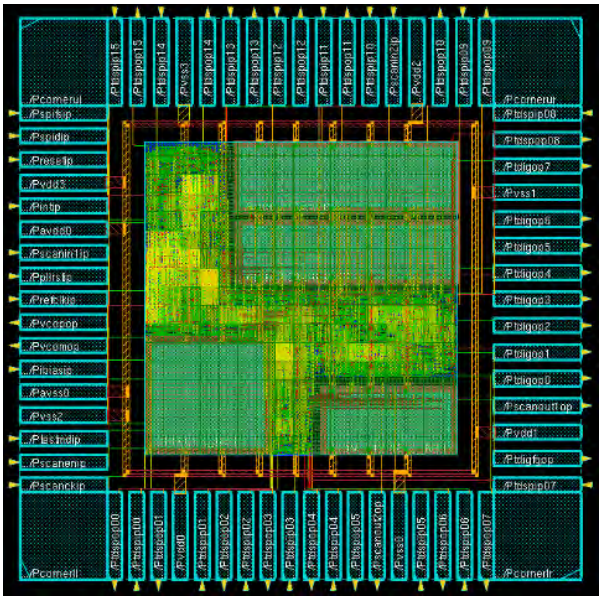


Figure 1.3: Placement Layout of Encounter of Cadence

Global Placement is divided into Analytical and Combinatorial. 1.4. Analytical Placement is a more promising field as it formulates the placement process using mathematical representation of both Linear and Non-Linear complexity. Solving such Linear equations,

is an easy, fast and robust way that all electronic circuit design and manufacturing companies, seek. However, Combinatorial Techniques give a precise and exact placement solution in relation to Analytical ones that give an approximation of the solution. Nevertheless, analytical's approximation solution is very close to the combinatorial's one.

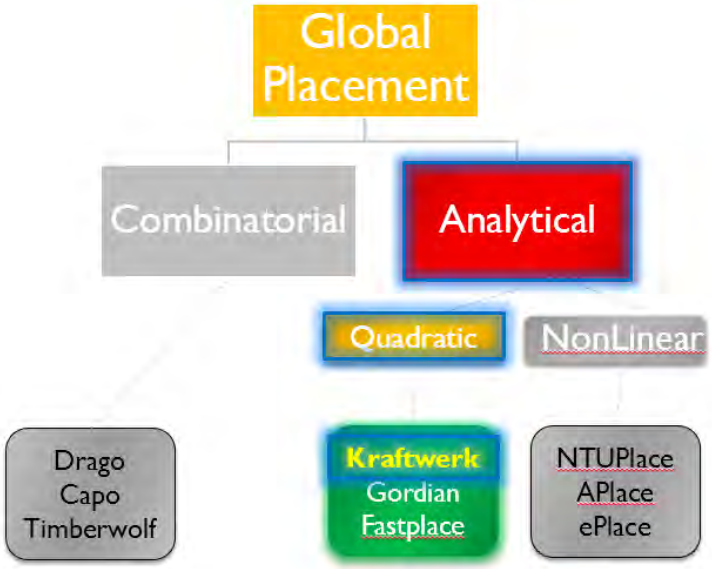


Figure 1.4: Placement Techniques

CHAPTER 2

KRAFTWERK2 ALGORITHM

Kraftwerk2 Algorithm is a fast Force-Directed Analytical Quadratic Placer. As described in the paper [7], it is formed by two major ideas. First, Kraftwerk is based on distributing the modules on the chip by using an additional force. The additional force is separated in two forces: the hold force and the move force. Both components are implemented in a systematic manner. This novel systematic force modeling yields the robustness of this powerful iterative placement algorithm resulting in an overlap-free placement. The second concept of Kraftwerk2 is the use of an exact linear net mode, the "Bound2Bound" (B2B) net model, which can be easily used in any other quadratic placer. This net model represents the half-perimeter wirelength (HPWL) in the quadratic cost function more efficiently as claimed. The HPWL in general is a linear metric for the net length and represents a common and efficient estimation for the routed wirelength.

2.1 Net Models

The main objective to placement problem is to reduce overlapping cells by simultaneously minimizing the total length of all nets. Connections between cells or pins or the above both are represented as two-pin connections, so as to express the wirelength. To achieve that, a simple net model is necessary. The most commonly used measurement of wirelength for any given net $e \epsilon E$, where E denotes the sum of the circuit nets, is the HPWL [1] and can be written as

$$HPWL(e) = \max_{i,j \epsilon e, i<j} \left| x_i - x_j \right| + \max_{i,j \epsilon e, i<j} \left| y_i - y_j \right| \tag{2.1}$$

The total wirelength is then given by the $\sum_{e \epsilon E} HPWL(e)$. Although HPWL is a convex

function, it cannot be directly minimized, because of the absolute distances $max\left|x_i - x_j\right|$ and $max\left|y_i - y_j\right|$, that convert the cost function to a non-differentiable one. Therefore, the solution lies on the weighted and square Euclidean distances between the two-pin connections of i.e. one net n, whose cost function Γ_n [5] is:

$$\Gamma_n = \sum_{e=(i,j)\epsilon E} \frac{w_{e,x}}{2}(x_i - x_j)^2 + \frac{w_{e,y}}{2}(y_i - y_j)^2 \tag{2.2}$$

where E is the set of two-pin connections that form net n. As a result, the total Quadratic Cost function is the

$$\Gamma = \sum_{n=1}^{N} \Gamma_n \tag{2.3}$$

In many years of research, many other types of net models were presented, each one suitable for different purpose and all of them featuring interesting pros and cons, which we will not analyze in this thesis. However, some of them are shown in the Figure 2.1 below.

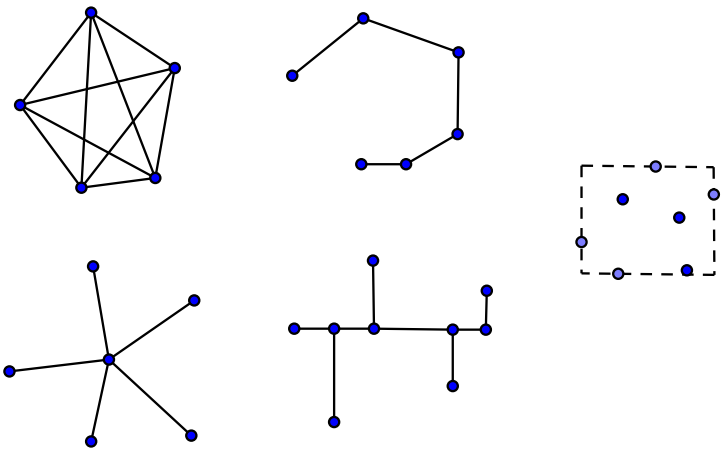


Figure 2.1: Net Models

2.1.1 Bound2Bound net model

In paper [7], a new, better as claimed and proved net model is described, the B2B net model. The new Bound2Bound net model is based on the idea of removing all inner two-pin connections, as presented in Figure 2.2, and to utilize only connections to the boundary pins. This net model is similar to the clique net model, but its connection

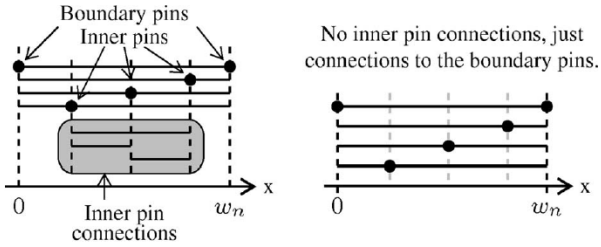


Figure 2.2: Inner Pins

weight W_{B2B} differs. If we focus on x-dimension, similarly for y-dimension, the B2B weight for one two-pin connection is:

$$w_{x,pq}^{\text{B2B}} = \begin{cases} 0, & \text{if pin } p \text{ and pin } q \text{ are inner pins} \\ \frac{2}{P-1}\frac{1}{\left|x_p^{\text{pin}} - x_q^{\text{pin}}\right|}, & \text{else.} \end{cases}$$

Figure 2.3: B2B net model weight value

In Figure 2.4, setting as boundary pins the $p = 1$ and $q = 2$ and given the $w_n = \left|x_1^{pin} - x_2^{pin}\right|$, we can write down the math we will conclude that the quadratic cost function of a net n is exactly the HPWL W_n, with zero approximation error.

$$\begin{aligned} \Gamma_{n,x} &= \frac{1}{2}\sum_{p=1}^{P}\sum_{q=p+1}^{P} w_{x,pq}^{\text{B2B}}\left(x_p^{\text{pin}} - x_q^{\text{pin}}\right)^2 \\ &= \frac{1}{2}\frac{2}{P-1}\left[\left|x_1^{\text{pin}} - x_2^{\text{pin}}\right| + \sum_{q=3}^{P}\left|x_1^{\text{pin}} - x_q^{\text{pin}}\right| \right. \\ &\qquad \left. + \sum_{q=3}^{P}\left|x_2^{\text{pin}} - x_q^{\text{pin}}\right|\right] \\ &= \frac{1}{P-1}\left[w_n + (P-2)w_n\right] \\ &= w_n. \end{aligned}$$

Figure 2.4: B2B Cost function

2.2 Quadratic Placement

Quadratic placement (QP) is formed by the rule that each net of the circuit is represented by two-pin connections. A circuit consists of three basic sets, a set of modules, a set of

pins and a set of two-pin connections, M, P, E, accordingly. All these sets are located inside the core area. Modules M are divided into movable and non-movable. Only the first ones are taken into account in the QP. In addition, pins P refer to the i/o pins of a chip and they are usually placed in locations close to their connections. Furthermore, two-pin connections E describe every connection between one and another module, one module and one pin or one and another pin. Pin to pin connections are rare in modern circuits. In order QP to be solved, i/o pins need to be placed first.

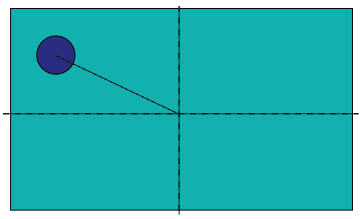


Figure 2.5: Inner Pin inside a module

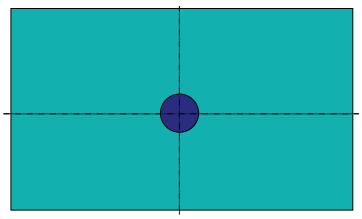


Figure 2.6: Centralized Inner Pin

Inside the modules, inner pins are presented as shown in Fgure 2.5 , so as to better visually understand the two-pin connections. But, in order to simplify things, an assumption is made that inner module pins are placed on the center of the module as shown in Fgure 2.6, and by referring to inner module pin connection we mean a connection to that module. In addition, a two-pin connection $e = (p,q)\ \epsilon\ E$ connects pin p with pin q, where p,q can be either i/o pins or inner module pins.

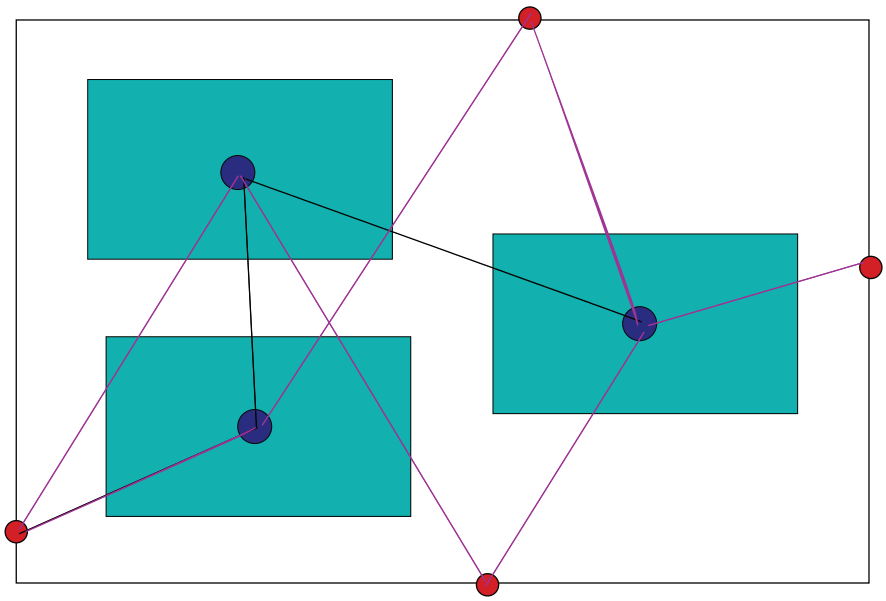


Figure 2.7: QP solution simple example

Gathering all these sets together, we can form the QP problem as shown in Figure 2.7. The big rectangle is the core area, the red dots are the pins that are placed on the boundaries of the core close to their connected modules and the blue dots represent the

inner pins, thus the module's connection point. Black and purple lines represent the two-pin connections module to module and module to i/o pin, respectively.

Our goal now is to minimize the cost function 2.3. In order to achieve this, we need to formulate the cost function in a matrix-vector notation so as to get the QP just by solving a linear equation $Ax = B$. Such formulation is shown below:

$$\Gamma_n = \frac{1}{2}x^T C_x x + \frac{1}{2}y^T C_y y + x^T d_x + y^T d_y + constant \tag{2.4}$$

Vectors x^T and y^T represent the movable modules' position. Matrices C_x and C_y represent all the two-pin connections. Vectors d_x and d_y refer to the connections between i/o pins and modules. By getting the first derivatives of the above equation 2.4 for x and y separately, then setting them to zero 2.5 2.6 and finally solving them w.r. to x and y, we end up with the QP solution, which are the locations of the modules inside the core area with the minimum wirelength.

$$\bigtriangledown_x \Gamma_n = C_x x + d_x = 0 \tag{2.5}$$

$$\bigtriangledown_y \Gamma_n = C_y y + d_y = 0 \tag{2.6}$$

The QP solution is usually the first step to Global Placement, and is characterized as Initial Placement. Having the components' location, all the connections and their connection weights, we just need a way to spread them, and in particular by using a force. If all these are combined, a new linear equation can be formed, analyzed later in section 2.4, whose solution will produce new updated locations in each iteration.

2.3 Krafwerk2 Forces

2.3.1 Net & Hold Force

In force-directed quadratic placement methods, two-pin connections can be translated into elastic springs. Thus, the set of two-pin connections, that represent a simple net model-set, create a spring system with energy equal to the quadratic cost function Γ 2.3. The derivative of this cost function represent a force that is called "net" force 2.7.

$$F_x^{net} = \bigtriangledown_x \Gamma_n = C_x x + d_x \tag{2.7}$$

Net Force is the product of the nabla positions of the components with the quadratic cost function. It is called net force, because this force is created by the two-pin connections, that represent the nets. Net force needs to be set to zero $F_x^{net} = 0$ in order to obtain an equilibrium state of the spring system. With just the net force, components tend to accumulate to the center of the core or in general to their initial location. Hence, an other force is needed to hold the equilibrium state and additionally spread them. In fact, this force is separated into two other forces, the "hold" force and the "move force".

$$F_x^{hold} = -(C_x x^{'} + d_x) \tag{2.8}$$

Hold Force is just an "inverse net force". This force is used to compensate the net force in the beginning of each placement iteration. It is formulated as show in 2.8. This force is using just the previous component's location in order to achieve a zero relocation, look 2.9. Components will stay at their current position without collapsing back to their initial one. Hence, no force accumulation is necessary, because each iteration is decoupled from the previous one and the algorithm can be restarted at any iteration. It is important to mention that hold is constant, as it is not affected by variable x inside the linear equation.

$$\begin{aligned} F_x^{net} + F_x^{hold} &= 0, \\ (C_x x + d_x) + (-(C_x x^{'} + d_x)) &= 0, \\ C_x(x - x^{'}) + d_x - d_x &= 0, \\ x - x^{'} &= 0, \\ \Delta_x &= 0 \end{aligned} \tag{2.9}$$

2.3.2 Move Force - Demand-Supply System

The third and major force of KW2 algorithm is the "move" force. Its major role is to spread cells (components) all over the core area, which is the main objective of global placement. Hence, module (component) overlap reduces until a specified limit, where the algorithm converges and stops. In order to represent the move force, a Distribution Model is formed and modeled by a Demand-Supply system.

Demand refers to the module overlap, whereas Supply to the placement area, usually the whole core. At the beginning and after Initial Placement e.g QP, the module overlap is really high. So, modules need to be moved to lower density regions, which is implemented with this Demand-Supply system that has to be balanced. The integral over the demand has to equal the integral over the supply, as presented in Figure 2.8. In addition, in the first iteration of KW2, $D^{sup} > D^{dem}$, because there is plenty of placement area for cells to be spread.

$$\iint_{-\infty}^{+\infty} D^{dem}(x,y)dxdy \;=\; \iint_{-\infty}^{+\infty} D^{sup}sup(x,y)dxdy$$

Figure 2.8: Demand-Supply system that needs to be balanced

In order to formulate this distribution system, a Rectangle function 2.9 is needed in order to point the positions that a cell occupies. x and y represent the locations inside the rectangle and x_{ll}, y_{ll} and w, h refer to the location and sizing of each cell, accordingly.

For each cell i, the Demand equals 2.10, whereas the Supply equals 2.11. $d_{cell,i}$ is usually set to 1, but it changes according to what kind of components we place i.e. mixed circuits with small and big cells, macro cells, etc. .

$$R(x, y, x_{ll}, y_{ll}, w, h) = \begin{cases} 1, & if\ 0 \le x - x_{ll} \le w \\ & OR\ \ 0 \le y - y_{ll} \le h \\ 0, & elsewhere \end{cases}$$

Figure 2.9: Rectangle Function of Demand-Supply system

$$D_{cell}^{dem}(x, y) = d_{cell,i} * R(x, y, x'_i - \frac{w_i}{2}, y'_i - \frac{h_i}{2}, w_i, h_i)$$

Figure 2.10: Demand value for each cell i

$$D_{cell}^{sup}(x, y) = d_{sup} * R(x, y, x_{chip}, y_{chip}, w_{chip}, h_{chip})$$

Figure 2.11: Supply value for each cell i

$$d_{sup} = \sum_{i=1}^{M+F} \frac{(d_{cell,i} * A_m)}{A_{chip}}$$

Figure 2.12: Supply density value for each cell i

On the other hand, d_{sup} has a constant value and is given by 2.12. It is the sum of all components' demand multiplied by each area and divided with the chip area, which means it is constant. This kind of distribution system can be easily translated in a more mathematical way as the electrostatic potential φ by the Poisson's equation 2.10.

$$(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}) * \varphi(x, y) = -D(x, y) \tag{2.10}$$

This differential equation's solution determines the target points $\dot{x}_i$ and $\dot{y}_i$. Target points are the positions that each cell is "forced" to move to in each iteration. They are determined, for a cell i by equation 2.11:

$$\dot{x}_i = x'_i - \frac{\partial}{\partial x}\varphi(x, y)|_{(x'_i, y'_i)} \tag{2.11}$$

Thus, the Gradients of Potential φ are collected in a vector Φ presented in Figure 2.13. The move force is determined by equation 2.12:

$$\boldsymbol{\Phi} = \left((\partial/\partial x)\varphi|_{(x'_1,y'_1)},\ (\partial/\partial x)\varphi|_{(x'_1,y'_1)},\ \ldots,\ (\partial/\partial x)\varphi|_{(x'_M,y'_M)} \right)^T$$

Figure 2.13: Gradient vector of Potential φ

$$\begin{aligned} F_{x,i}^{move} &= \dot{w}_i * (x_i - \dot{x}_i), \\ F_{x,i}^{move} &= \dot{w}_i * (x_i - (x_i^{'} + \Phi), \\ F_{x,i}^{move} &= \dot{w}_i * ((x_i - x_i^{'}) + \Phi) \\ F_{x,i}^{move} &= \dot{w}_i * (\Delta_x + \Phi) \end{aligned} \tag{2.12}$$

where Δ_x is the cell movement in one iteration and $\dot{w}_i$ is the move force strength translated into a weight value. This weight value is fundamental for the KW2 functionality as it affects the distance that a cell is moved. After having presented all the core force-directed behaviour of KW2, it is high time we move to the "Core System", the Linear Equation.

2.4 Krafwerk2 Linear Equation

From QP, the Connection Laplacian Matrix C is ready and obviously needs to be part of the linear equation and specifically on the left hand. In addition, the gradients of move force stored in vector Φ need to be integrated. Except from the gradients, inside move force's formulation, a weight value is determined for every cell, too. These weights are stored in a diagonal matrix $\dot{C}$. The KW2 core linear equations for both x and y dimensions is 2.13:

$$(C_x + \dot{C}_x)\Delta x = -\dot{C}_x\Phi_x \quad \text{and} \quad (C_y + \dot{C}_y)\Delta y = -\dot{C}_y\Phi_y \tag{2.13}$$

In more detail Connection Matrix C_x is similar to C_y. Focusing on x direction we form it in the following way:

1. We form the **Adjacency Matrix** of our design, which is a diagonally zero valued matrix with weights or 1 in every connection between components.
2. We form the **Pin Connection Matrix** that is similar to the Adjacency but is particularly for connection between i/o pins and components
3. We then form the **Degree Matrix** that is a diagonal matrix with value, the sum of every row.

An example from book [4], a connection graph design, like the way netlists are translated before placement, is presented in Figure 2.14. Adjacency matrix, Pin matrix and Degree matrix are showm in 2.15, 2.16, 2.17, accordingly.

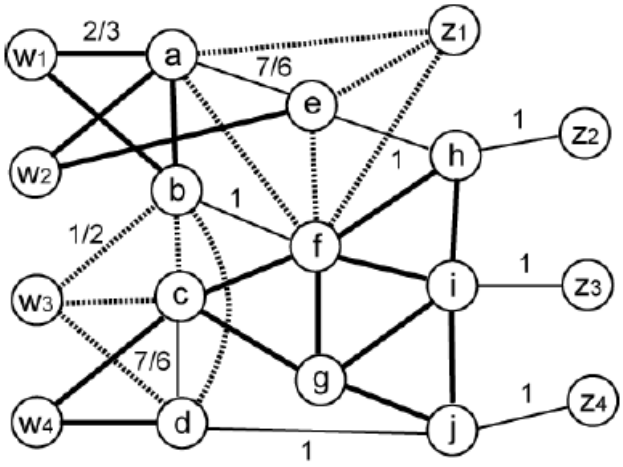


Figure 2.14: Graph Design

$$\begin{pmatrix}
0 & \frac{2}{3} & 0 & 0 & \frac{7}{6} & \frac{1}{2} & 0 & 0 & 0 & 0 \\
\frac{2}{3} & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{2} & 0 & \frac{7}{6} & 0 & \frac{2}{3} & \frac{2}{3} & 0 & 0 & 0 \\
0 & \frac{1}{2} & \frac{7}{6} & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\frac{7}{6} & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 1 & 0 & 0 \\
\frac{1}{2} & 1 & \frac{2}{3} & 0 & \frac{1}{2} & 0 & \frac{2}{3} & \frac{2}{3} & \frac{2}{3} & 0 \\
0 & 0 & \frac{2}{3} & 0 & 0 & \frac{2}{3} & 0 & 0 & \frac{2}{3} & \frac{2}{3} \\
0 & 0 & 0 & 0 & 1 & \frac{2}{3} & 0 & 0 & \frac{2}{3} & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{2}{3} & \frac{2}{3} & \frac{2}{3} & 0 & \frac{2}{3} \\
0 & 0 & 0 & 1 & 0 & 0 & \frac{2}{3} & 0 & \frac{2}{3} & 0
\end{pmatrix}$$

Figure 2.15: Adjacency Matrix

The result is the Laplacian matrix 2.18, the connection one like C_x and is formed from the subtraction of Degree minus the Adjacency.

The other matrix $\dot{C}$ derives from the weights of move force and is initialized in the setup stage of KW2. In addition, it reforms in every iteration in the quality control stage, analyzed in section 2.5. Gradient results from the Poisson's equation and directly probed to the linear solver.

At last, the final form of the KW2's Linear Equation 2.13 is a really interesting procedure as described in Figure 2.19. In order to form the linear system, all forces need to be simultaneously set to zero. By adding them all together, as defined in section 2.3, we have:

In a "magical" but totally mathematical way, we formed the basic and more complex

$$\begin{pmatrix} \frac{2}{3} & \frac{2}{3} & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\ \frac{2}{3} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{2}{3} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{2}{3} & 0 & 0 & 0 & 0 \\ 0 & \frac{2}{3} & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 2.16: Pin Connection Matrix

$$\begin{pmatrix} \frac{25}{6} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{23}{6} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{25}{6} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{23}{6} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{23}{6} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{31}{6} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{8}{3} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{10}{3} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{11}{3} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{10}{3} \end{pmatrix}$$

Figure 2.17: Degree Matrix

linear system, which has three major degrees of freedom, that should be described in detail.

1. Weight Matrix $\dot{C}$
2. Gradient Φ
3. Net Model Weights inside C Matrix

$$\begin{pmatrix}
\frac{25}{6} & -\frac{2}{3} & 0 & 0 & -\frac{7}{6} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\
-\frac{2}{3} & \frac{23}{6} & -\frac{1}{2} & -\frac{1}{2} & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & -\frac{1}{2} & \frac{25}{6} & -\frac{7}{6} & 0 & -\frac{2}{3} & -\frac{2}{3} & 0 & 0 & 0 \\
0 & -\frac{1}{2} & -\frac{7}{6} & \frac{23}{6} & 0 & 0 & 0 & 0 & 0 & -1 \\
-\frac{7}{6} & 0 & 0 & 0 & \frac{23}{6} & -\frac{1}{2} & 0 & -1 & 0 & 0 \\
-\frac{1}{2} & -1 & -\frac{2}{3} & 0 & -\frac{1}{2} & \frac{31}{6} & -\frac{2}{3} & -\frac{2}{3} & -\frac{2}{3} & 0 \\
0 & 0 & -\frac{2}{3} & 0 & 0 & -\frac{2}{3} & \frac{8}{3} & 0 & -\frac{2}{3} & -\frac{2}{3} \\
0 & 0 & 0 & 0 & -1 & -\frac{2}{3} & 0 & \frac{10}{3} & -\frac{2}{3} & 0 \\
0 & 0 & 0 & 0 & 0 & -\frac{2}{3} & -\frac{2}{3} & -\frac{2}{3} & \frac{11}{3} & -\frac{2}{3} \\
0 & 0 & 0 & -1 & 0 & 0 & -\frac{2}{3} & 0 & -\frac{2}{3} & \frac{10}{3}
\end{pmatrix}$$

Figure 2.18: Laplacian Matrix C_x

$$\boldsymbol{F_x^{net} + F_x^{hold} + F_x^{move} = 0}$$

new position old position target position

$$F_x^{net} + F_x^{hold} = -F_x^{move}$$

$$(C_x * x + dx) + \left(-(C_x * x' + dx)\right) = -\dot{C}_x * (x - \dot{x})$$

$$C_x * x + dx - C_x * x' - dx = -\dot{C}_x * ((x - x') + \Phi_x)$$

$$C_x * x - C_x * x' = -\dot{C}_x * ((x - x') + \Phi_x)$$

$$C_x * (x - x') = -\dot{C}_x * ((x - x') + \Phi_x)$$

$$C_x * \Delta x = -\dot{C}_x * (\Delta x + \Phi_x)$$

$$C_x * \Delta x + \dot{C}_x * \Delta x = -\dot{C}_x * \Phi_x$$

$$\boldsymbol{(C_x + \dot{C}_x) * \Delta x = -\dot{C}_x * \Phi_x}$$

Figure 2.19: Linear Equation Formulation C_x

2.5 Quality Control

In order to obtain a better spreading, paper's [7] authors propose a weight assignment for a module i with the value $\dot{w}_i = \frac{a_{modi}}{A_{avg}} \frac{1}{M}$, where M is the number of movable components inside the core area, A_{modi} and A_{avg} refer to the area of component i and the total average components' area , respectively. The goal of this formulation is to achieve a proportional weight value - move force - according to the module's dimensions. Hence, large modules tend to spread further in each iteration.

It is correctly claimed that, high quality placement need more CPU time and *vice*

versa. This trade-off between quality and execution time is controlled by the user by setting a target movement μ_τ. In each iteration, the module movement μ is compared to the target movement and the weight w_i adapts its value in the quality control stage. The function that changes these weights is a bounded hyperbolic tangent 2.14. If $\mu > \mu_\tau$ then $\kappa > 1$, whereas $\kappa = 1$ when $\mu < \mu_\tau$. In Figures 2.20 and 2.21 we can see the way the execution time and the HPWL "moves" in this Paretto diagram. Finally, we observe the values of κ according to the movement μ.

$$\kappa(\mu) = 1 + tanh(ln(\mu_\tau/\mu)) \tag{2.14}$$

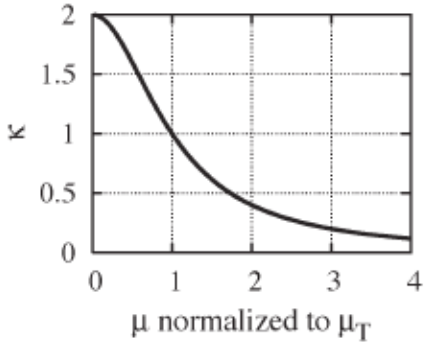


Figure 2.20: Scale factor κ depending on μ and μ_τ

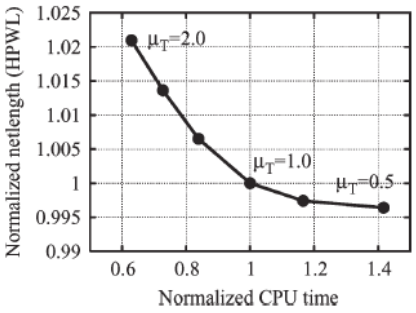


Figure 2.21: Trade-off between execution time and placement quality

2.6 Kraftwerk2 Algorithm Flow

Kraftwerk2 tries uses analytical methods. It needs an Initial Placement as input so as to iteratively reach an optimal solution or the Global Placement, as it is characterized. With Initial Placement cells are placed inside the core area, which can be done either by placing them in random x and y coordinates, or by getting the Quadratic Problem (QP) solution. The QP is the minimized solution produced by solving a linear system. That kind of linear system is formed using the weighted connections between cells and pins and the coordinates of the i/o pins, whose solution results in the optimal locations of every cell inside the core area, with the minimum Total Wirelength (TWL).

The first stage of Kraftwerk2 Global Placement is the Initial Placement. Then, in each iteration, the algorithm needs to solve a more complex linear system for each dimension, as shown in Equation 2.13. Simultaneously, KW2 creates all the matrices and vectors needed, in order to form the linear equation. Both this iterative solving technique and a Quality Control System help Kraftwerk2 achieve a solution with significantly reduced cell overlap. The pseudocode of the basic and most important algorithmic steps is presented below. Then, in Figure **??** we present you the Algorithm's Flow Design.

INITIAL *Placement {...}*
GLOBAL *Placement*
begin
  while *moduleOverlap* $< 20\%$ **do**
    Create Demand-Supply system $D(x,y)$
    Calculate Potential $\Phi(x,y)$
    Apply B2B net-model
    for *x-direction (analogously for y-direction)* **do**
      Create $C_x, \dot{C}_x, \Phi_x$
      Solve (2.3) w.r.t Δ_x
      Update module position x by Δ_x
    end
    Quality Control
  end
end
DETAILED *Placement { ... }*

Algorithm 1: Kraftwerk2 algorithm

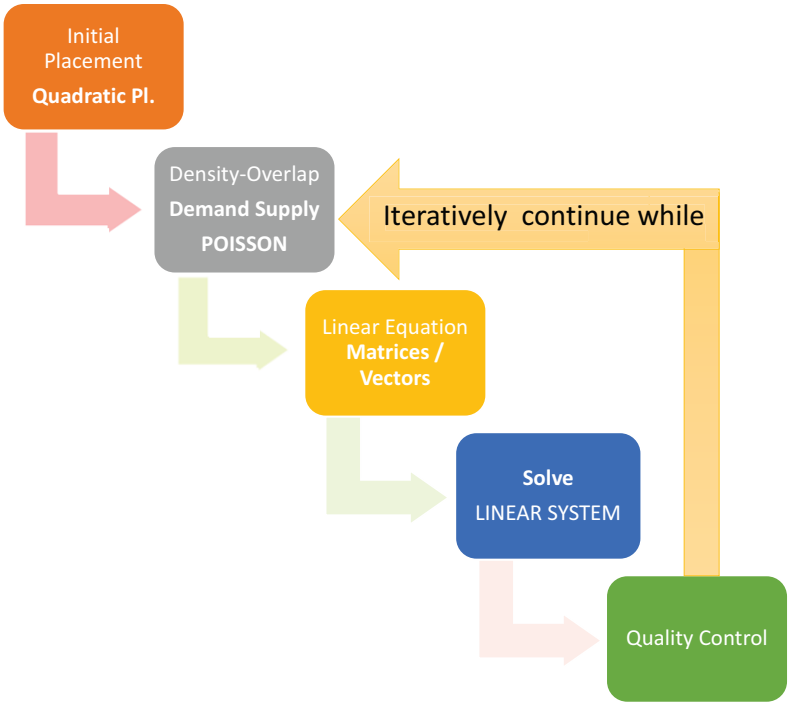


Figure 2.22: KW2 FLow Design

2.7 Thesis Motivation and Purpose

After this Chapter's 2 further analysis, we concluded that KW2 is an interesting, easily parametrized and fast analytical placement technique. It is also very important to note that KW2 is fully Linear in relation to other placement algorithms that are NonLinear and really slow, like `NTUPlace`, `mPL`, `APlace`, etc. Its Linear complexity in both the linear equation system and in the Poisson's Distribution system was the basic stimulus to start our further analysis and exploration.

In addition, with KW2's three degrees of freedom, numerous ways of quality optimization appear. We implement and extend the algorithm to an updated version, named KW2++, that introduces a slightly different weight approach, an other Demand-Supply system formulation and a novel quality bounded control. Thus, our KW2++ rejects incorrect and invalid solutions in each iteration, where KW2 wasn't. By accepting all movements, three major things happen. The algorithm converges either very slow or never, much more than 25 iterations, as claimed in the KW2 paper [7], are needed and of course the solution is totally incorrect. We optimized it in order converge extremely fast, and enriched its Features, ike clustering support, etc. , as shown in Figure 2.23. We met various test cases that KW2 couldn't handle, and solved them in our updated version.

Feature		KW2	KW2+
Bins	Continuous	✓	✓
	Discrete	✗	✓
Weight Approach	(Acell/Aavg) * (1/M) (very fast)	✓	✓
	(Acell/Aavg) (very fast)	✗	✓
Termination Condition	Overlap	✓	✓
	Max Density	✗	✓
Quality Control	tahn(log(1/x))	✓	✓
	1 / x	✗	✓
Upper Bound	Bounce Effect	✗	✓
	Spreading Artifacts	✗	✓
Distribution Modeling	Poisson	✓	✓
	Gaussian Blur	✗	✓
Solver Support		DIMEPACK / Unknown	GSL / CSPARSE / UMFPACK / MKL
Clustering Support		✗	✓

Figure 2.23: KW2 and KW2++ Feature comparison

Moreover, it should also be mentioned that, the paper's [7] proposal does slightly change the form of the connection matrix by assigning weights, in this already formed Connection Matrix, that derive from the B2B net model. The B2B net model's weights slightly change the Matrix form, resulting in no actual solution's quality optimization. Thus, we propose the simplest approach Point2Point, by representing all connections as two-pin connections.

CHAPTER 3

KRAFTWERK2++ IMPLEMENTATION AND ANALYSIS

In this chapter, a detailed analysis of the implementation of our Kraftwerk2++ algorithm is presented, which is the main concept of this thesis. Furthermore, each step of the algorithm will be technically analyzed and discussed, by mentioning various problems encountered throughout the whole process and the way they were successfully solved.

Although, analytical algorithms, like Kraftwerk2, use simple mathematical solving techniques, they can be difficult to implement. Through the design process, various solving and coding techniques were tested, targeting solving speed, execution time, complexity, scalability and correctness of the result. Thus, the workload was divided in two parts.

The first part was assigned to me, including the whole algorithm flow and the Linear System formulation and solving. The second part was assigned to Michalis Giaourtas, a colleague of mine, which was targeting the formulation of the Demand-Supply System, a very demanding and paramount part of the algorithm.

3.1 Solver Options

Solving Linear Equations is a mathematical process that can be accomplished in many different ways. A huge variety of direct or iterative methods exist like the Cramer's Rule, Gaussian Elimination, LU Decomposition, Gauss Jordan Elimination, Crout's Method [2] and many others. Fortunately, these methods are being optimized and integrated into Mathematical Solvers. Due to the fact that the development was done in C programming language, the focus was on compatible solving packages, some of which are MUMPS, MOSEK, GSL, LAPACKE, CSPARSE, UMFPACK. After continuous testing the first successful solvers were the GSL and Lapacke. Although they were fast, they couldn't handle and support large matrices. Thus, the solution was to try sparse matrix solvers

like the rest three mentioned above. In this section, a detailed Linear Solver analysis will be done, which was a fundamental step for the completion of this thesis.

3.1.1 GSL

The GNU Scientific Library (GSL) is a numerical library in C programming language. It is a free software under the GNU General Public License. The library provides a wide range of mathematical routines one of which is Linear Algebra Operations. Let's assume that we want to solve the linear system $Ax = B$ with A matrix and B vectors that are already formed, which is not a time-demanding process. The source code is very simple and presented below. The core of this solving technique are the two functions:

```
// ********** Start Function Definition ********** //
int gsl_linalg_LU_decomp(gsl_matrix * A, gsl_permutation * p, int *
    signum);
int gsl_linalg_LU_solve(const gsl_matrix * LU, const
   gsl_permutation * p, const gsl_vector * b, gsl_vector * x);
// ********** end Function Definition ********** //

// ********** Start Solving Process ********** //
// allocate solution vector X_gsl for #modules =
   totalindexedcomponents//
X_gsl = gsl_vector_alloc(totalindexedcomponents);

// allocate permutation matrix p //
p = gsl_permutation_alloc(totalindexedcomponents);

// solve using LU decomposition //
gsl_linalg_LU_decomp(A_gsl, p, &s);
gsl_linalg_LU_solve(A_gsl, p, b_gsl, X_gsl);

// clear permutation p //
gsl_permutation_free(p);

// clear solution //
gsl_vector_free(X_gsl);
// ********** End Solving Process ********** //
```

In spite of being such a simple method just by performing an LU decomposition and solving the linear system, it cannot handle bigger designs, meaning big matrices, and it crashes. In addition, the above technique has to be used twice for both x and y dimensions that leads to two times more execution time. Moreover, this method requires too much resources in memory, which means that it is poorly memory optimized. In the end, it is a great and quick method for small designs, GNU licensed and is the reason why we started with this simple implementation. Some experiments and results will be mentioned in 4.

3.1.2 LAPACKE

After GSL inability to serve our purposes in larger net-lists, an other package helped us to expand our experiment's range. LAPACKE is a C interface for LAPACK which is an older, FORTRAN-based and efficient math library. Is is under BSD free License. By using the 3.1.1 subsection's example, but for both x and y dimension, we now need only one routine to solve the linear system, by using LU decomposition and assuming that we have already created the connection matrix A and the module to i/o pin's location vector. Source code is presented below.

```
// ********** Start Function Definition ********** //
lapack_int LAPACKE_dsysv(int matrix_layout, char uplo, lapack_int n
    , lapack_int nrhs, double * a, lapack_int lda, lapack_int * ipiv
    , double * b, lapack_int ldb)
// ********** end Function Definition ********** //

// ********** Start Solving Process ********** //
// Solve for both x and y dimension //
info = LAPACKE_dsysv(LAPACK_ROW_MAJOR,'U' , totalindexedcomponents,
    DIMNUM, A_lap, totalindexedcomponents, ipiv, b_lap, DIMNUM);

// Check for the exact singularity //
if (info > 0)
  {
    printf("The diagonal element of the triangular factor of C,\n")
        ;
    printf("U(%i,%i) is zero, so that C is singular;\n", info, info
        );
    printf("the solution could not be computed.\n");
    exit (2);
  }

// Solution is stored in b_lap vector //

// ********** End Solving Process ********** //
```

As we can observe, it is such a simple method and really direct and effective, as can solve more than one linear system simultaneously. $DIMNUM$ defines the number of dimensions that we want to solve, which in our case is just two. Also, B vector gets the solution of the linear system. Compared to the GSL method, LAPACKE handles both memory and speed in a more enhanced way. Thus, it helped us solve and test more design netlists. Nevertheless, LAPACKE started lagging and stalling in much bigger test-cases, until it was became slow and finally crashed due to lack of memory. Although it was a breakthrough and time-consuming task, we started searching for more efficient and memory conservative techniques. as presented in the next subsections.

3.1.3 SUITESPARSE - Sparse Matrix Solving Techniques

In this section, sparse solving method are presented using the mathematical packages CSPARSE and UMFPACK. These packages support solvingof linear equations in com-

pressed column and triplet sparse format. Thus, the challenge lied to the conversion of the above A matrix to a sparse form.

3.1.3.1 CSPARSE

CSPARSE is a concise sparse Cholesky factorization package included in the SuiteSparse Suite. In needs a compressed-column input in order to solve the linear equation. The solving flow is also so simple and is presented below. The tough part of this part of the thesis, was the conversion of the connection matrix into sparse format. A part of my colleague's Angelina Delacura Master Thesis, was the formulation of the QP. With the help and guidance of Associate Professor Christos Sotiriou, they achieved to convert QP connection Matrix into triplet format.

```
// ********** Start Solving Process ********** //
// allocate temporary SuiteSparse/CXSparse sumbolic and numeric
   temporary variables //
S = NULL;
N = NULL;
S = cs_dl_malloc(totalindexedcomponents, sizeof(cs_dls));
N = cs_dl_malloc(totalindexedcomponents, sizeof(cs_dln));

// allocate global solution vector  //
X_cs = cs_dl_malloc(totalindexedcomponents, sizeof(double));

// create compressed matrix compressed_A_cs from A, where A is in
   triplet //
compressed_A_cs = cs_dl_compress(A_triplet);

// remove duplicate entries from compressed matrices //
int removedsuccess = cs_dl_dupl(compressed_A_cs);

if (removedsuccess == 0)
{
  printf("ERROR: SCS Could not Remove Duplicate Entries from
     Compressed Matrix.\n");
}

// solve linear sparse system //
S = cs_dl_sqr(2, compressed_A_cs, 0); // order = 2, qr = 0 //
N = cs_dl_lu(compressed_A_cs, S, 1); // tol = 1 //
cs_dl_ipvec(N->pinv, B_cs, X_cs, totalindexedcomponents);

cs_dl_lsolve(N->L, X_cs);
cs_dl_usolve(N->U, X_cs);
cs_dl_ipvec(S->q, X_cs, B_cs, totalindexedcomponents);

// clear compressed matrix compressed_A_cs //
cs_dl_free(compressed_A_cs);

// ********** End Solving Process ********** //
```

So simple as that, this method performed amazingly faster than the previous ones, in spite of the fact that it needed two times to execute for x and y direction. It could handle almost all designs, but after testing it was lagging and being so slow in much bigger designs. Results will be analyzed on chapter 4.

3.1.3.2 UMFPACK

Fortunately, Suitesparse has numerous solving methods, one of which is UMFPACK, which is a multifrontal LU factorization technique. In other words, it performs LU decomposition on a more optimized way like right memory management, error handling, *etc.* leading to better solving speed and ability to cope with all given netlists with approximately 1 million components, wchich are tested so far. It consists of similar routines as csparse, like triplet to compressed format conversion, symbolic and numeric formulation and a solving routine. It solves one dimension in a routines' flow pass/execution. Below, the short example of $Ax = B$ is presented.

```
// ********** Start Solving Process ********** //

// allocate global vector X_umf //
X_umf = malloc(totalindexedcomponents * sizeof(double));

// allocate temporary SuiteSparse/UMFPACK compressed column
   temporary vectors //
Ap = NULL; Ai = NULL; Ax = NULL;
Ap = (SuiteSparse_long *) malloc((totalindexedcomponents + 1) *
   sizeof(SuiteSparse_long));
Ai = (SuiteSparse_long *) malloc(A_triplet->nz * sizeof(
   SuiteSparse_long));
Ax = (double *) malloc(A_triplet->nz * sizeof(double));

// create compressed column arrays Ap, Ai, Ax //
status = umfpack_dl_triplet_to_col(totalindexedcomponents,
   totalindexedcomponents, A_triplet->nz, A_triplet->i, A_triplet->
   p, A_triplet->x, Ap, Ai, Ax, NULL);

if (status != UMFPACK_OK)
    {
      printf("ERROR: UMFPACK Reports Compressed Column Conversion
         Error for the x Direction!\n");
      switch (status)
        {
          case UMFPACK_ERROR_out_of_memory:
                {
                  printf("ERROR: UMFPACK Reports Out of Memory
                     Error!\n");
                  return 0;
                  break;
                }
          case UMFPACK_WARNING_singular_matrix:
                {
```

```
                    printf("WARNING: UMFPACK Reports Singular Matrix
                        !\n");
                    break;
                }
            }
        }

// create symbolic factorisation temporary data //
status = umfpack_dl_symbolic(totalindexedcomponents,
    totalindexedcomponents, Ap, Ai, Ax, &Symbolic,
    UMFPACK_STRATEGY_AUTO, NULL);

if (status != UMFPACK_OK)
    {
        // ... ABOVE ERROR CHECKING ... //
    }

// create numeric factorisation temporary data, based on symbolic
    result //
status = umfpack_dl_numeric(Ap, Ai, Ax, Symbolic, &Numeric,
    UMFPACK_STRATEGY_AUTO, NULL);

if (status != UMFPACK_OK)
    {
        // ... ABOVE ERROR CHECKING ... //
    }

// free symbolic factorisation temporary data //
umfpack_dl_free_symbolic(&Symbolic);

// solve in X_umf, using temporary numeric factorisation data //
status = umfpack_dl_solve(UMFPACK_A, Ap, Ai, Ax, X_umf, B_umf,
    Numeric, UMFPACK_STRATEGY_AUTO, NULL); // sys = UMFPACK_A (Ax =
    b) //

if (status != UMFPACK_OK)
    {
        // ... ABOVE ERROR CHECKING ... //
    }

// free numeric factorisation temporary data //
umfpack_dl_free_numeric(&Numeric);

// free temporary compressed column arrays //
free(Ap); free(Ai); free(Ax);

// ********** End Solving Process ********** //
```

All the memory optimization and error handling is performed inside these routines. For the Kraftwerk's purposes, only memory increase and matrix singularity were the only, so far, warnings and errors that occurred and perfectly handled. This part of the thesis, was the most demanding, time consuming and really hard. Hours of experimentation and

testing led to an optimal solving technique. Therefore, being able to solve every sparse linear system efficiently, the execution time of KW's solving steps in each iteration was extremely decreased, and minimised, due to a code trick optimization performed, which will be discussed later in section 3.3.

3.2 Our Demand Supply System

The Demand Supply System was the most important and difficult part to implement due to its high complexity. Michalis Giaourtas created this system, using bins formulation. Using Poisson equation he was able to determine the Gradients for the components needed for the right part of the Linear System. Figure 3.1 presents a plot of the distribution of the components in a core area, the Poisson's result and a sample of the Gradients created.

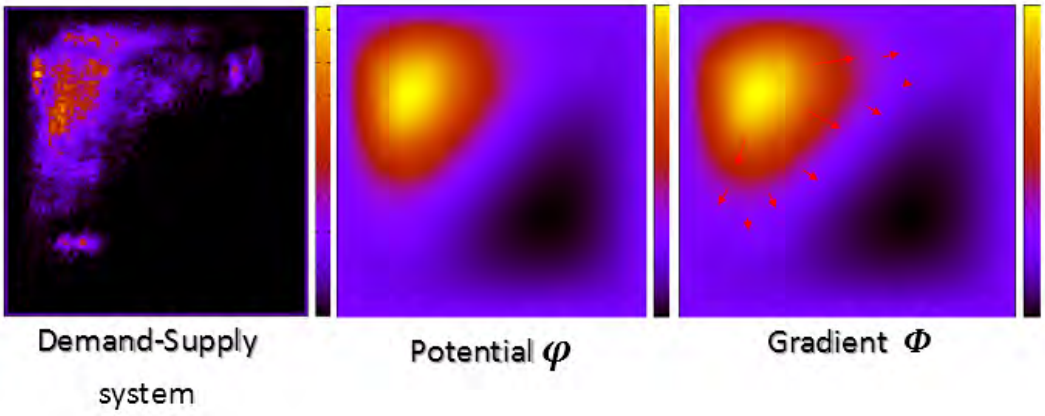


Figure 3.1: Demand Supply System

3.3 Linear System Analysis

This section focuses on the technical details, coding techniques and implementation of the Kraftwerk's core. In the experimentation process, many different implementations were proposed and completed, due to the various solver options. In this section, though, only the final optimized one that uses sparse formulation, is being mentioned. As described in chapter 2, the linear system is composed by two parts, the left and the right, which are being analyzed below.

3.3.1 Linear Equation Left part

Left segment of the linear equation consists of the connection matrix C and the weight matrix $\dot{C}$. Laplacian matrix C, is symmetric, diagonally positive and off-diagonally either negative or of zero value. It is mostly sparse and already formed from Angelina's QP

project. Both the QP and the KW need the same matrix, as the connections between components and i/o pins are not altered, through placement. Moreover, weight matrix $\dot{C}$ is diagonal positive and off-diagonal zero valued.

On the one hand, matrix C is formed twice for each dimension x and y, but is the same. On the other hand, diagonal weight matrix $\dot{C}$ is is formed also twice, but it differs in each dimension due to the iterative quality control that is being analyzed later in section 3.3. Thus, in KW2 Setup phase, the above matrices are formed, and only the $\dot{C}$ is reformed in every iteration of KW2 Core phase.

By being more precise, we needed to perform a matrix addition. Due the zero valued off-diagonal $\dot{C}$, only the diagonal entries were added, iteratively one by one with the ones of connection matrix C. A sparse seek matrix function was available in SuiteSparse package, but it was too slow, because it was searching all entries to reach one. Thus, we implemented an trick with a pre-phase mode inside the KW2 Setup. With a matrix complete pass, we stored the diagonal entries locations and stored them in an new vector. Then, every time, we needed to alter the diagonal values, we were just using the already stored locations. In total, the execution time reduction was really paramount. Below, this simple coding trick is presented.

```
k ← 0 for c ← 1 to NonZeroCmatrixValues do
    /* if diagonal value found, store location                    */
    if C.i[c] == C.j[c] then
        DiagonalLocation(k) = c;
        k++;
    end
    /* if all diagonal locations are discovered, then exit        */
    if NonZeroCvalues == TotalComponents then
        exit1;
    end
end
```

Algorithm 2: Seek Sparse Diagonal Entries trick

The weight matrix can control the algorithm behaviour, and that is why is one of the quality control "ingredients". With the right value handling in every iteration, we can achieve the best placement quality, in execution time and wirelength, see section 3.3.

3.3.2 Linear Equation Right part

On the other part of the equation, we meet two crucial parameters. One is once again the weight matrix $\dot{C}$ and the other one is the Gradient of the distribution system proposed in section 3.2. The math is so simple, that just by performing a multiplication, we instantly formulate the right hand of the equation. Nevertheless, through quality control, we try to affect these values in a novel way, which will be described in section 3.3.

3.4 Quality Control

Our KW2++ algorithm introduces a slightly different weight approach 3.1, as presented in detail in section 3.4. The paper's implementation (KW2) was working really good, but it was spreading the components either completely wrong creating *Spreading Artifacts* or extremely slow. In order to solve slow execution, we actually removed the fraction $\frac{1}{totalcomponents}$ by assigning bigger weights in each component in matrix $\dot{C}$. However, we managed to spread the cells in a quicker way, but we couldn't handle bigger and dense (high-overlapping) designs. Components were bounding all over the core area in each iteration. In order to solve that, we formed an Upper Bound function to restrict the Algorithms behaviour by limiting the average movement of each component in every iteration. Hence, not only we fixed the Algorithm's Convergence, but we vanished the *Spreading Artifacts* phenomenon.

$$\dot{w}_i = \frac{a_{modi}}{A_{avg}} \tag{3.1}$$

3.4.1 Spreading Artifacts Effect

Meanwhile, we noticed that KW2 paper's algorithm, couldn't handle high region overlapping designs. Components' weights in KW2 are really small in value resulting in tiny movements, preserving the high density, being grouped and being unable to converge at all, as proved in the experiments in Chapter 4. We named this behaviour *"Spreading Artifacts Effect"*. We firstly observed it in `cordic` design and then in `b19` and `des_perf` designs as shown in Figures 3.2 and 3.3 , accordingly.

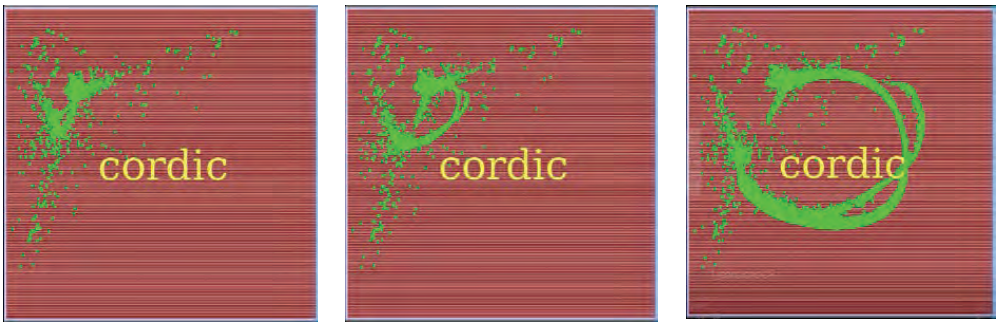


Figure 3.2: Spreading Artifacts Effect in KW2 (solved in KW2++) *cordic* design

Of course, this phenomenon was successfully solved using the Upper Bound (UB) solution of KW2++,as displayed below in Section 3.4.2.

3.4.2 Bouncing effect and our UB solution

In Figure 3.4 we notice this bouncing behavior over the core area. It is caused by the big weights assigned in every component, which affects the component's locations x and y.

Figure 3.3: Spreading Artifacts Effect in KW2 (solved in KW2++) in *b19* and *des_perf* design

The target points were in incorrect positions inside or usually outside of the core area. This phenomenon is present in high-dense designs and could also appear in KW2 by just assigning bigger or lower values either in component's weights or in the component's gradient in each iteration, referring to the three ways of freedom mentioned before.

The solution was simple enough, just by limiting the movement of the components Fortunately, we successfully solved it by the Upper Bound (UB) method as already introduced and tested it on one of the biggest and of high overlap designs `cordic` and `b19`, accordingly. The result was expected as show in Figures 3.5 and 3.6, analogously. To sum up, our Upper Bound method does not only solve the bouncing effect but many other phenomenon, too.

Rejecting invalid movements is the novel part of our KW2++ and my major contribution. From now on, KW2++ is able to handle every transformation in whichever part of the linear equation, meaning the three ways of freedom. Thus, our new KW2++ surpasses the original KW2 algorithm.

3.5 Different number of Bins

In KW2++, bins play a paramount role. Large number of bins result in a more detailed spreading, whereas less bins tend to either group components or slightly spread them.

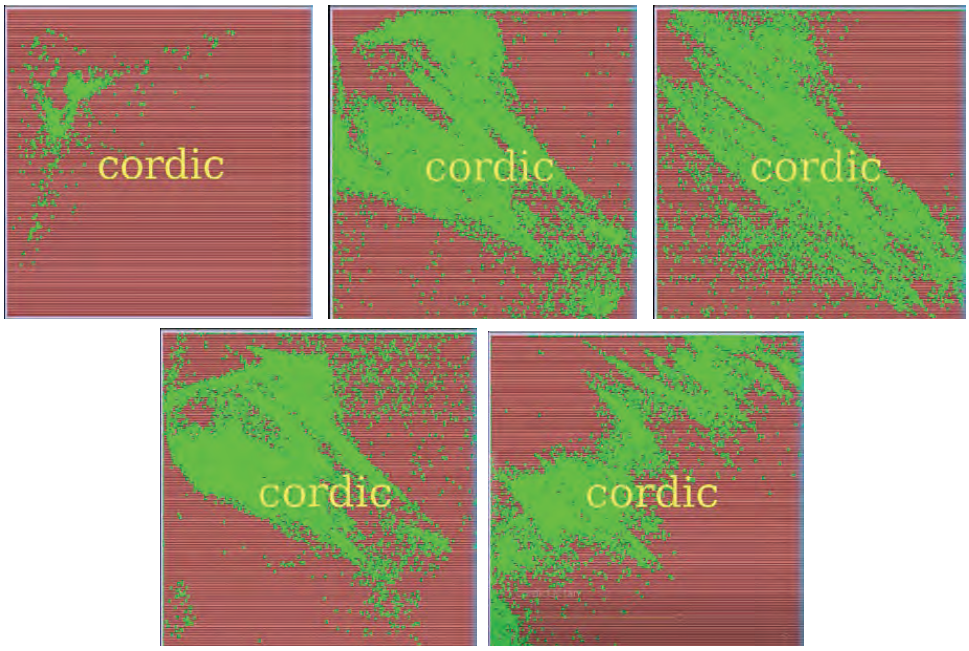


Figure 3.4: Bouncing Effect in *cordic* design

For instance, on the left part of figure 3.7 we can observe that design `fft` spreads a little with a bin grid 20x20, whereas on the left part of figure 3.8 `b19` is somehow grouping components in the center of each bin. Of course, KW2++ is based on the best bin formulation by setting the size of the bin to the *site width*, which is the smallest measurement unit inside the core. With such bin size, KW2++ algorithm succeeds in spreading the components correctly, right parts of the above mentioned figures, by preserving the minimum wirelength, which will be analyzed shortly. Simultaneously, due to the solving capability of the Sparse Linear Solver *UMFPACK*, the increase of bins does not affect the solving speed and of course the fast Convergence.

3.6 Aspect Ratio

An other experiment was performed to verify the algorithm correct execution for one more time. By changing Aspect Ratio, the core shape is being altered from a rectangle to a quadrangle. This alteration in the core sizing was not mentioned and possibly not handled in KW2, but fully supported in KW2++. Experiments in figure 3.9, of "BCM2" prove that our griding system and Demand-Supply system is able to handle such strange but possible scenarios. In this example, aspect ration 2 is used in the left part and aspect ration 0.5 in the right part. In fact, KW2++ could successfully handle bigger and lower aspect ratio values, too.

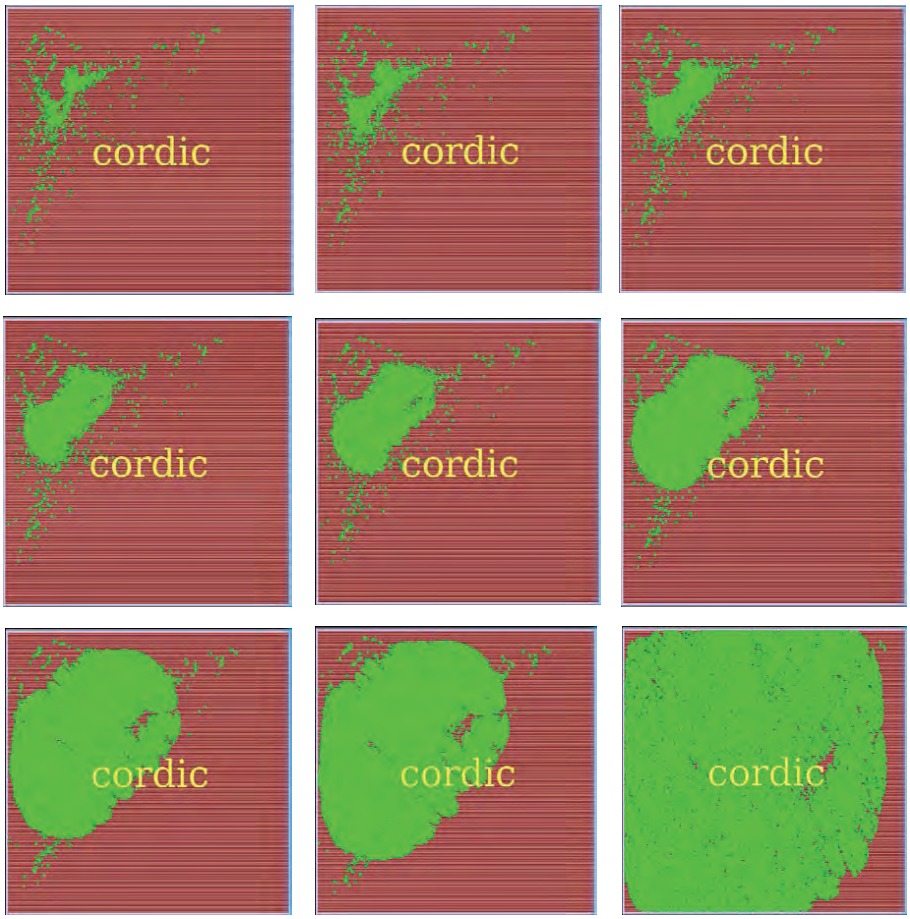


Figure 3.5: Upper Bound vanishing Bouncing Effect in *cordic*

3.7 Center-placed Components

In order to cope with every possible initial placement, we experimented on a specific corner case, in which all components are placed in a specific location inside the core. As the component's location, we set the center of the core. We noticed that, in small designs, where components are less in number and one cell can occupy more core space than in bigger designs, cell spreading was acting strangely. In particular, on the upper left part of figure 3.10, we observe an initial placement of all components of "BCM1" in the center of the core. After setting the bins' number in much smaller than the default value, we came up with the strange results on the right upper and lower part of figure 3.10. Then, we revert the bins' number in its initial default value and everything worked perfectly (look left bottom part of the figure). Hence, we once again experimentally proved and concluded that the way bins are formed is really important in the functionality of KraftWerk2 algorithm. Of course, we packed this feature in our enhanced KW2++ algorithm.

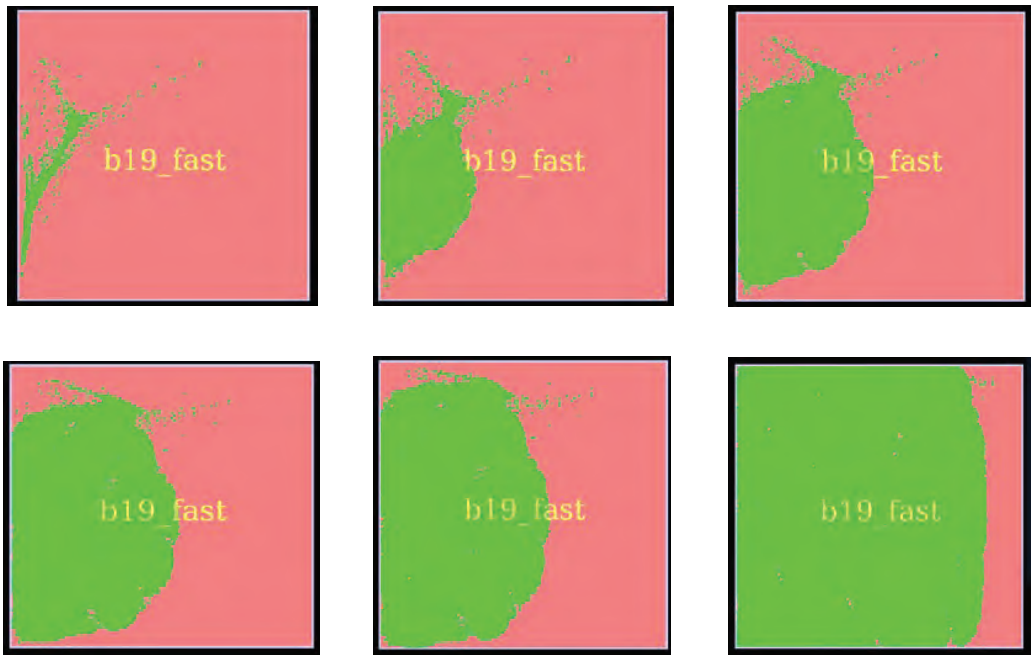


Figure 3.6: Upper Bound vanishing Bouncing Effect in *b19*

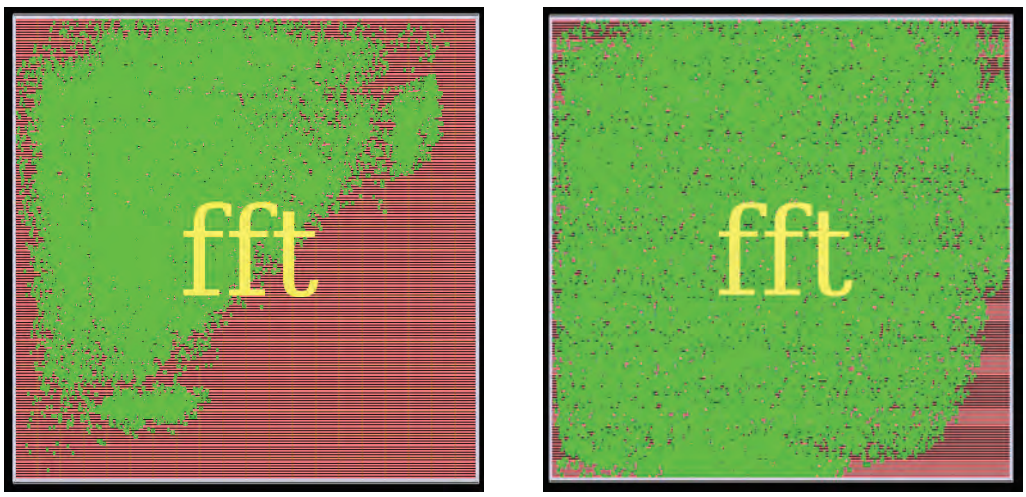


Figure 3.7: Spreading Artifacts Effect in KW2(right) and its solution in KW2+(left) in *fft* design

3.8 Clustered Support

Using our EDA Tool clustering support formulation, we slightly transformed the code to work with both components and clusters. Figure 3.11 is a first working attempt of our KW2++ algorithm to handle clusters. Of course, this version is in an early alpha stage and looks very promising.

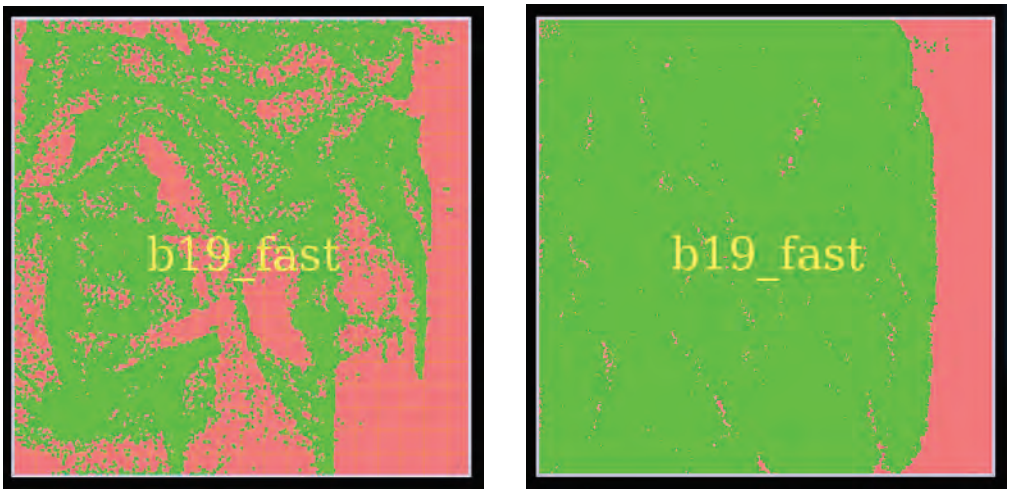


Figure 3.8: Spreading Artifacts Effect in KW2(right) and its solution in KW2++(left) in *b19* design

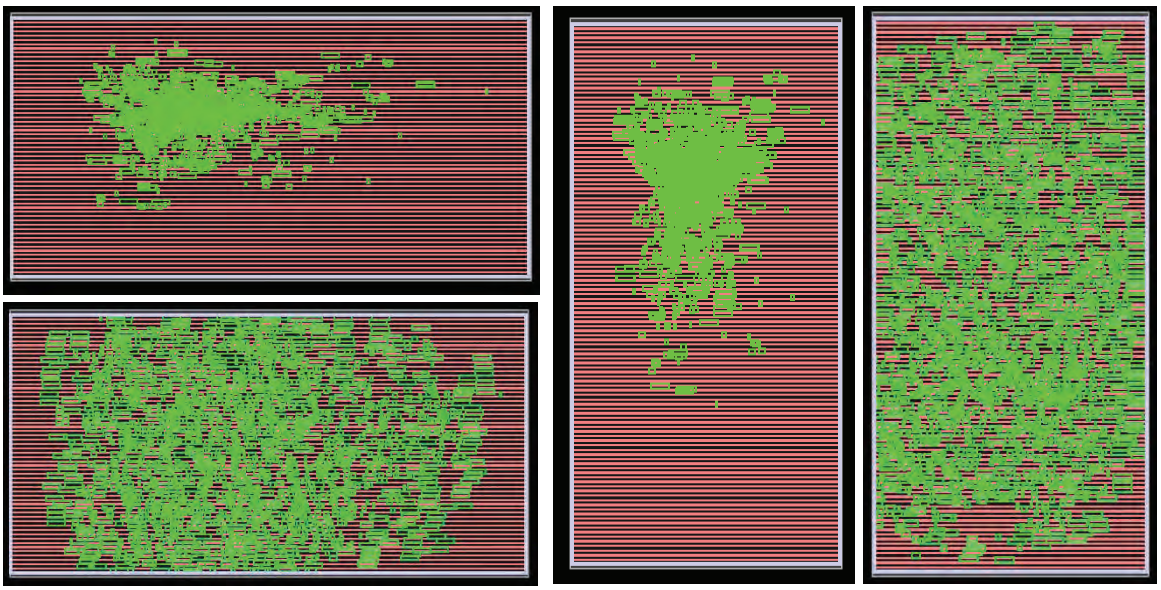


Figure 3.9: Aspect Ration Support

3.9 Interface

Our KW2++ Algorithm has a TCL interface where specific variables that affect the execution result can be set. The Tcl Option List is the following:

- **[-solver < 1 | 2 >]**
  - CSPARSE solver
  - UMFPACK solver

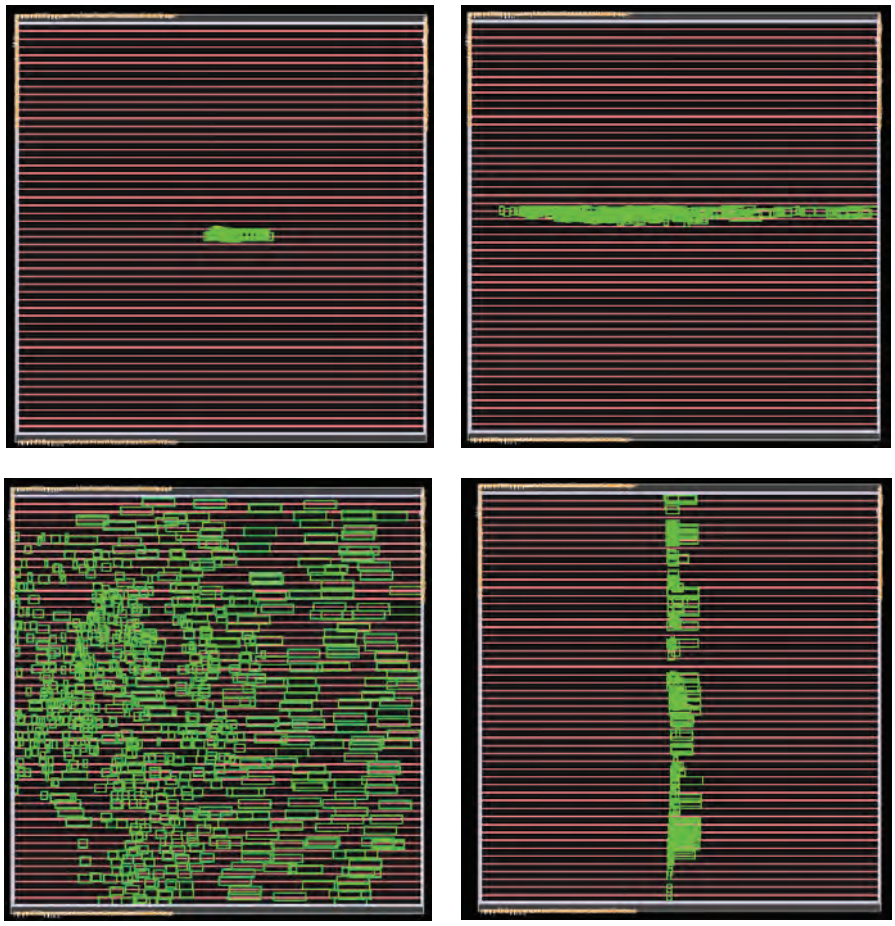


Figure 3.10: Center Placement Support

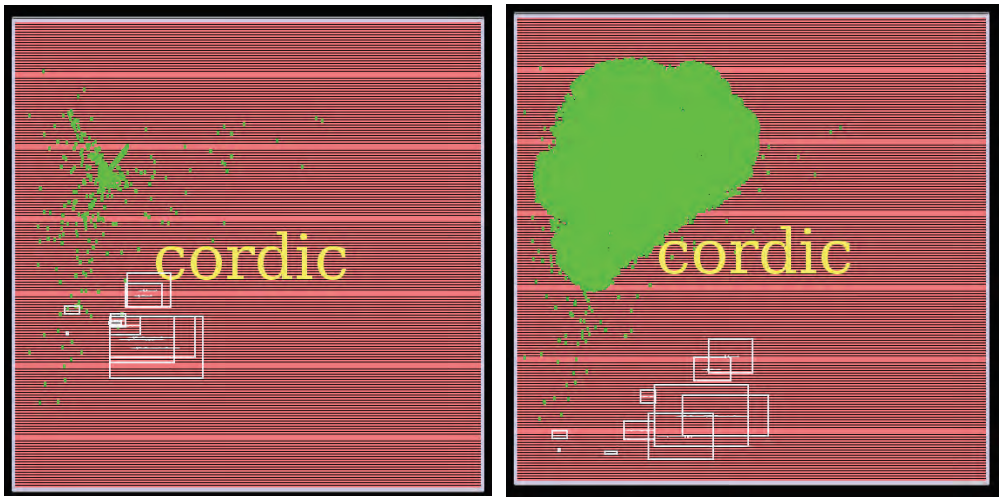


Figure 3.11: KW2++ on *fft* design supporting clusters.

- **[-poisson < 0 | 1 | 2 >]**
  - Poisson Method
  - Gaussian Blur Method
  - Fast Gaussian Blur Method
- **[-weightmethod < 0 | 1 >]**
  - (Acell/Avg)*(1/numberOfComponents) paper [7] approach
  - (Acell/Avg) our approach
- **[-qc < 0 | 1 | 2 >]**
  - Without Quality Control
  - With QC by Hyperbolic Tangent function (paper [7] method)
  - 1/x Michalis Giaourtas proposed function
- **[-iterations < iterations >]**
  - Number of maximum iteration
- **[-terminate < O | T | M >]**
  - terminate by *Overlap*
  - terminate by *Total Density*
  - terminate by *Maximum Overlap*
  - Either wise terminate on the first-meet termination condition

Many other sub-functions were created to control, termination condition, maximum density and overlap value, bins setting and drawing, all helping a better execution experience and flexibility.

CHAPTER 4

KRAFTWERK EXPERIMENTS RESULTS

This thesis chapter is the most crucial, as our KW2++ algorithm proves, via numerous experiments, its placement quality by handling various placement issues, by creating optimization techniques and improving the algorithm flow, compared to the paper's one KW2. More that 15 circuits were tested in different case scenarios, but only ten are presented using graphs, figures and tables. All the benchmarks mentioned above are mostly Academic ones form ICCAD and ISPD. In table 4.1 these circuits are presented with their characteristics that are needed in this chapter. The number of components of the designs range from 700 to almost 800,000. The smallest three are hierarchical designs, but as mentioned in previous sections, only flat netlists are supported in this implementation.

In the first section 4.1 of this chapter, we demonstrate and prove the best solving

Design Name	**Num. Components**	**Type**
Bench1	716	Hierarchical
Bench2	2,346	Hierarchical
Bench3	18,796	Hierarchical
bridge32_1	30,675	Flat
fft	32,281	Flat
cordic_I4	41,601	Flat
des_perf_1	112,644	Flat
matrix_mult	155,325	Flat
b19	219,268	Flat
leon3	649,191	Flat
leon2	794,286	Flat

Table 4.1: My benchmarks

package used, whereas in the rest we explain and justify our algorithmic extension and its novel characteristics via targeted and numerous experiments.

??:

Bins Number Gradient Method	**Utilization**	**Aspect Ratio**	**Gradient Method**
Default 20 %	50%	1.0	Poisson

4.1 Solver Capability

Fast linear equation solving could not be achieved if deep experimentation and testing was not performed. Hence, below we present you some interesting results. QP is the initial placement and of course just a solution of a linear equation system. Kraftwerk2 needs to solve one or more linear equations per iteration, too. Thus, at the beginning we mention results of the QP solution.

The hardware specifications of the machinery used for all experimentation process is:

CPU: 12-core Intel Xeon
RAM: 50Gbytes available

To begin with, in figure 4.1 the solving summary of the four basic linear solver is presented. As we can observe Sparse Techniques managed to accomplish the solving process, due to the less memory and system resources needed. The other solvers, were failing during the solving, basically due to the demanding large amount of memory needed. However, they could possibly have handled the other benchmarks, but they would still need a huge period of execution time, which was an issue for us, too. It should be noted that UMFPACK "beats" the rest of the solvers as presented in figure 4.1 both in solving, resources handling and basically in execution time. Thus, it was the best choice for us.

4.1.1 Execution Time for QP solution

In this experiment, we compare GSL to LAPACKE, then the CSPARSE to UMFPACK solvers and finally all together. On the one hand, in figure 4.2 we can notice that GSL is really slow when we try to solve a pretty small, in number of components, design and it finally crashes due to lack of memory. For instance, "Bridge" benchmark, as presented in figure 4.3, is being solved by UMFPACK in just **1 second**, but using GSL t takes **524 minutes**.

The difference is enormous, and the design is still one of the small ones. On the other hand, LAPACKE needs much less time, but quite much memory to solve some larger designs, but it exceeds the given resources, too.

Design ↓ Solver →	GSL	LAPACKE	CSPARSE	UMFPACK
BECM1	√	√	√	√
BECM2	√	√	√	√
BECM3	√	√	√	√
bridge32_1	√	√	√	√
fft		√	√	√
cordic_l4		√	√	√
des_perf_1			√	√
matrix_mult			√	√
b19			√	√
leon3				√
leon2				√

Figure 4.1: Net Models

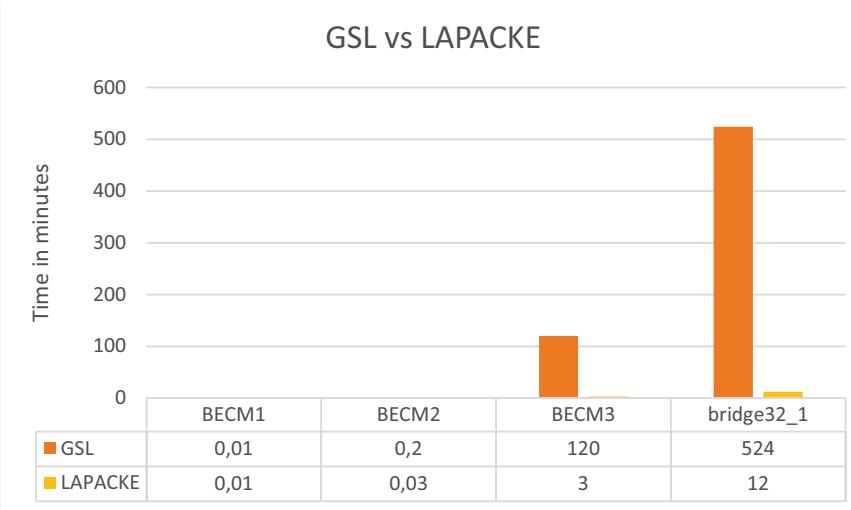


Figure 4.2: GSL vs LAPACKE
0

Furthermore, in figure 4.4 CSPARSE can be easily reach UMFPACK's execution time, but when we exceed 50,000 components, then the time exponentially rises. The result are very interesting and also cryptic. Some bigger designs can be solved in less time than some smaller ones. This is happening, due to the form of the connection matrix. If more components and i/o pins are connected, then the matrix or sparse matrix becomes bigger.

To conclude, UMFPACK experimentally is proven as the most efficient method for fast solving of really large linear systems. Hence, that is the reason that we chose it over the other options. In figure 4.5 we can see the results of a quadratic placement (QP), meaning the solution of a single linear system.

It is high time, we go through and present you some pictures of the QP solution

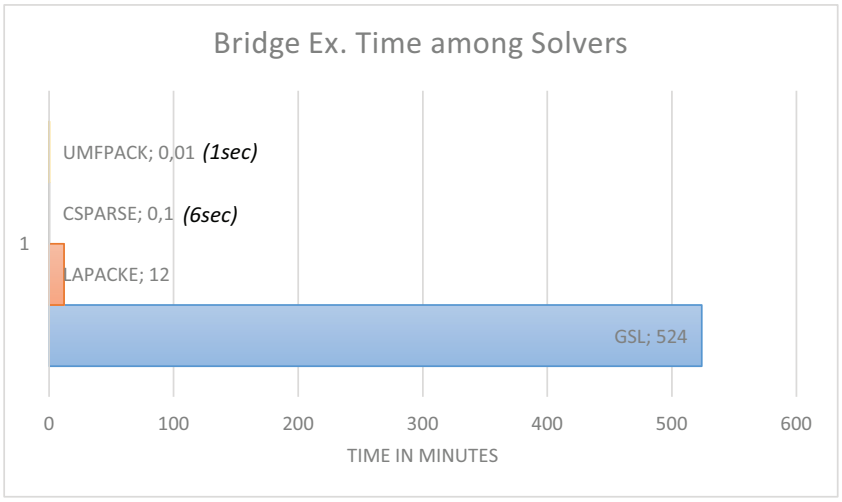
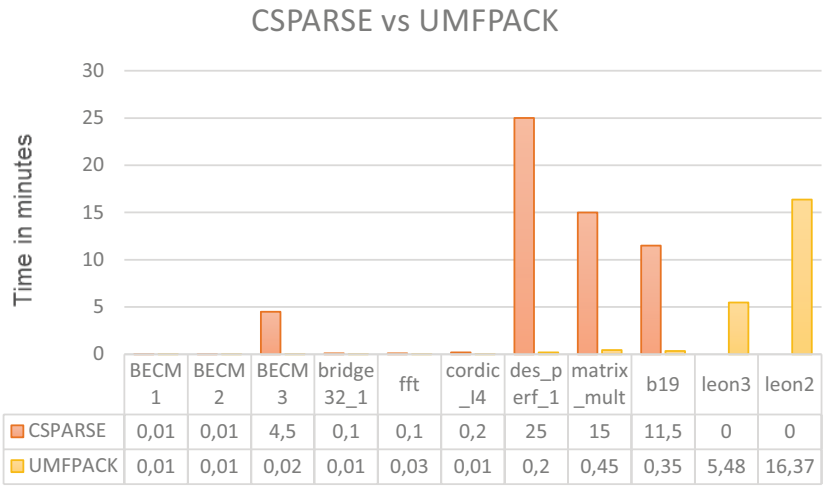


Figure 4.3: Bridge over the solvers



	BECM 1	BECM 2	BECM 3	bridge 32_1	fft	cordic _I4	des_p erf_1	matrix _mult	b19	leon3	leon2
CSPARSE	0,01	0,01	4,5	0,1	0,1	0,2	25	15	11,5	0	0
UMFPACK	0,01	0,01	0,02	0,01	0,03	0,01	0,2	0,45	0,35	5,48	16,37

Figure 4.4: CSPARSE vs UMFPACK

of the above ten designs. In the left column we can see both the connections between components (blue lines) and connections between components and i/o pins (orange lines), also called fly-lines. The components are colored green. On the right column we can only see the second connections mentioned above.

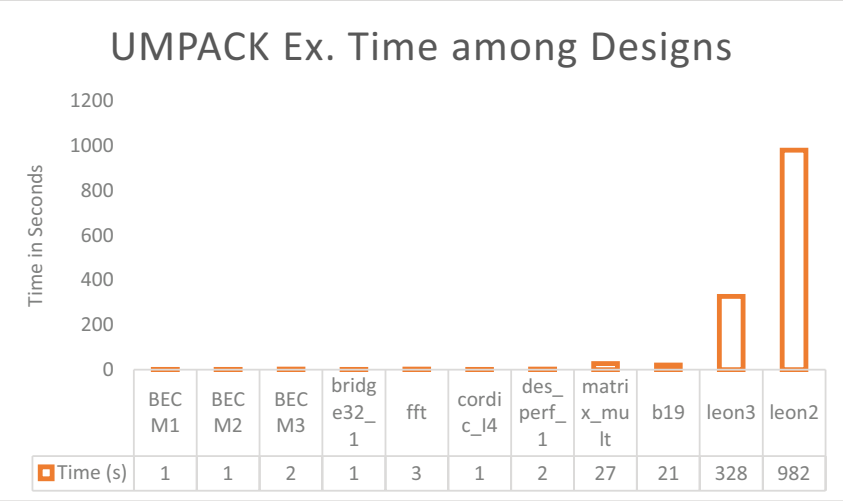
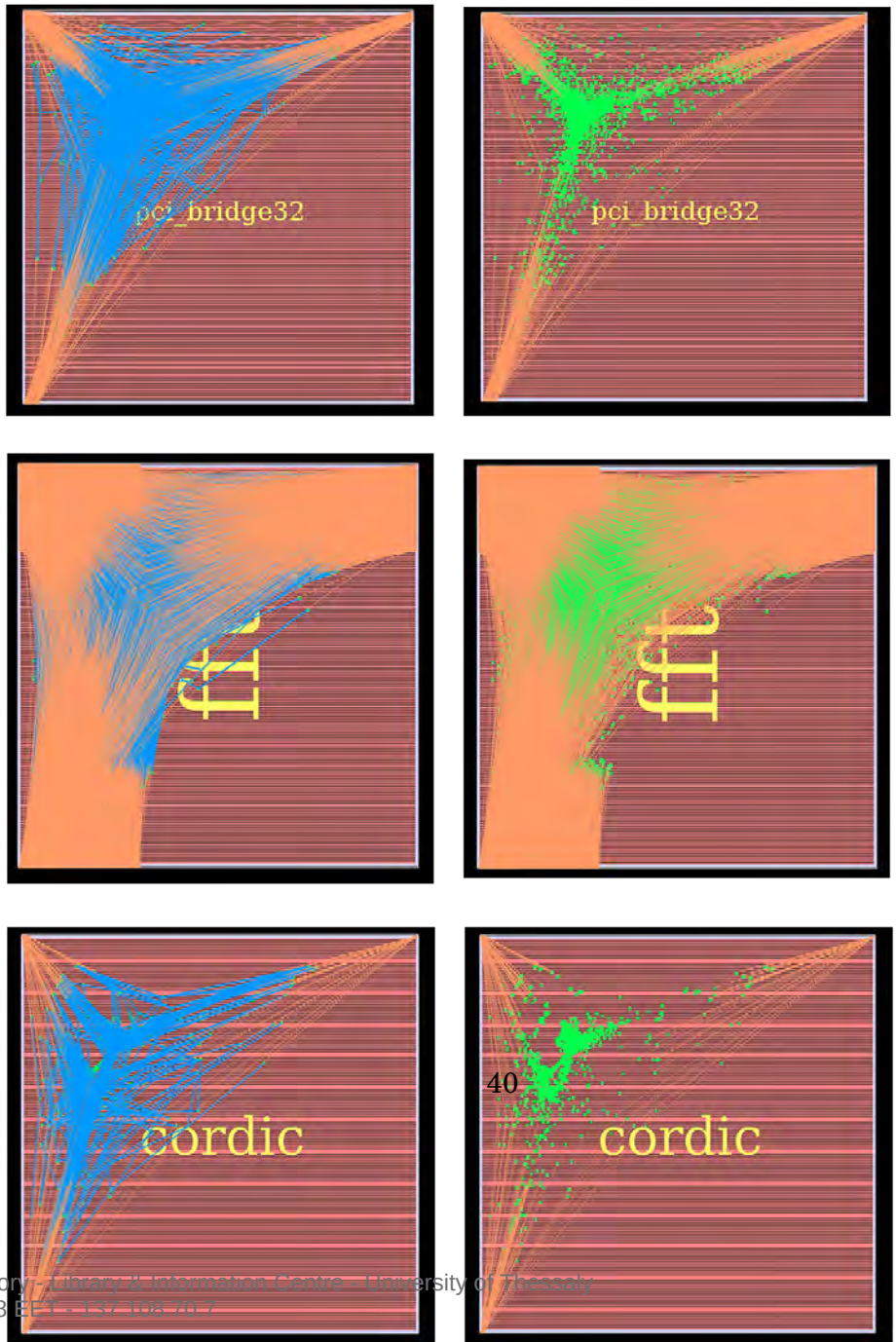
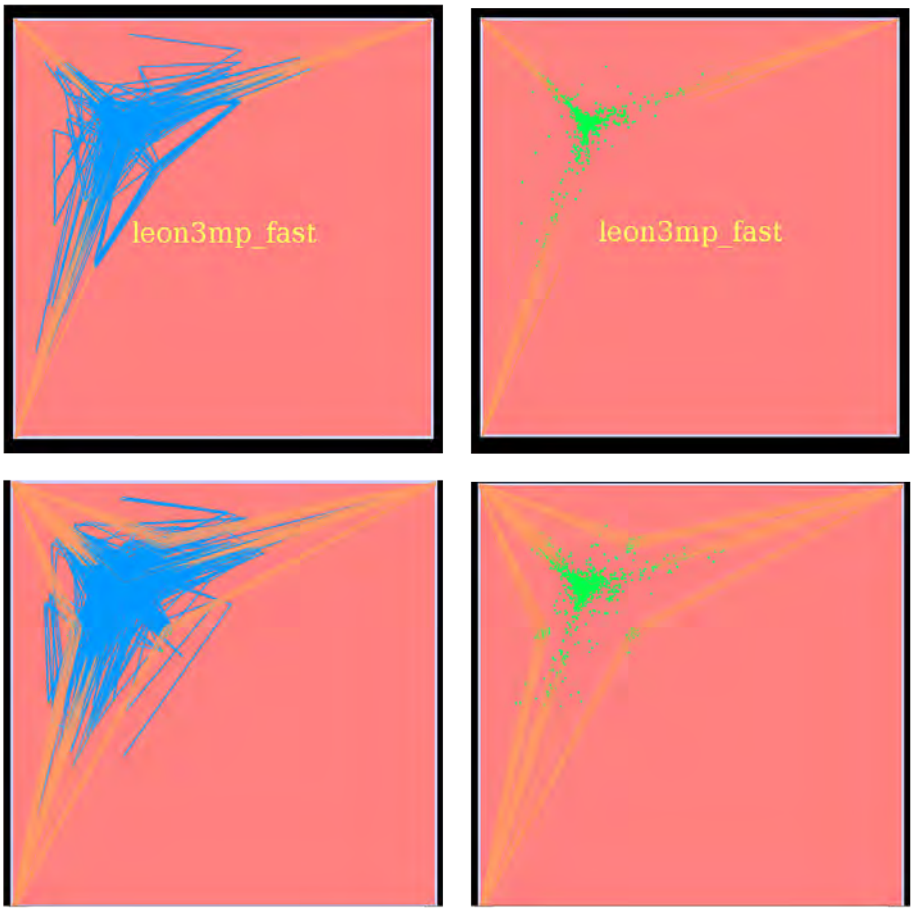


Figure 4.5: UMFPACK execution time of QP over the ten benchmarks given





4.2 Kw2 vs KW2++

After forming an idea of how fast and efficiently Quadratic Placement can be solved, let us have a better look on the comparison between the original KW2 and our new KW2++. By starting from the correct solving capability, meaning the ability to handle the designs correctly.

We can notice in Figure 4.6 that in the original KW2 cannot solve in a correct way six out of eleven designs tested, by creating these *Spreading Artifacts* and not be able to converge. Hence, in this section we will present you some result regarding just the first 5 benchmarks.

In Figure 4.7 we observe that KW2 is slightly better in Wirelength than our KW2++. This is obvious as we assign bigger weights in each component, resulting to a bigger, but still bounded movement. It should be noticed again that there is a trade-off between spreading quality and fast execution time. KW2++ is really fast, as show in Figure 4.8, which means that having such a really small difference in Wirelength, we can assume that KW2++ is much better in both ways, achieving the best *Average* solution.

Design	Algorithm	
	KW2	KW2++
BECM1	√	√
BECM2	√	√
BECM3	√	√
bridge32_1	√	√
fft	√	√
cordic_I4	IS NOT SUPPORTED	√
des_perf_1	IS NOT SUPPORTED	√
matrix_mult	IS NOT SUPPORTED	√
b19	IS NOT SUPPORTED	√
leon3	IS NOT SUPPORTED	√
leon2	IS NOT SUPPORTED	√

Figure 4.6: KW2 vs KW2++ support design list

Table 4.2: Wirelength comparison between KW2 nad KW2++

Design Name	KW2	KW2++
BCM1	35771,675	36885,125
BCM2	150803,363	166518,255
BCM3	1224389,631	1563070,047
bridge32_1	1597310,517	2615599
fft	8266159,983	8335183,294
cordic_I4	3149038,739	3170817,087

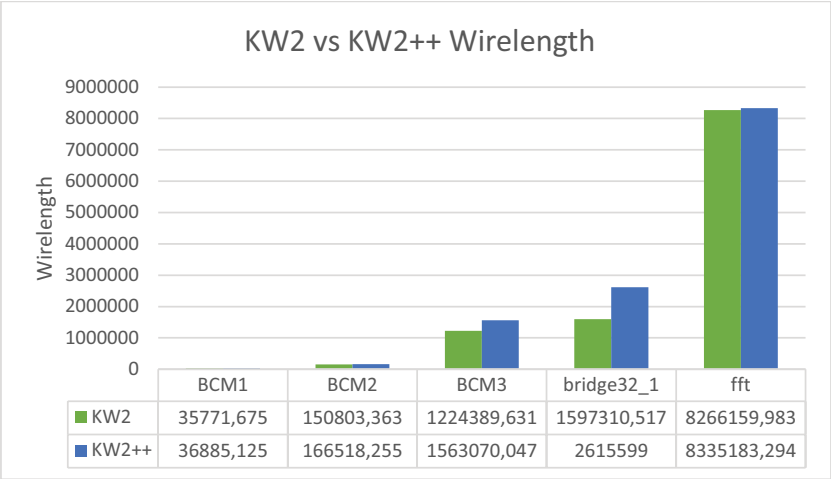


Figure 4.7: KW2 vs KW2++ wirelength comparison

Table 4.3: Execution Time comparison between KW2 and KW2++

Design Name	Time KW2(s)	Time KW2++ (s)
BCM1	31	2
BCM2	480	3
BCM3	64560	144
bridge32_1	38580	296
fft	115200	184
cordic_I4	136800	165

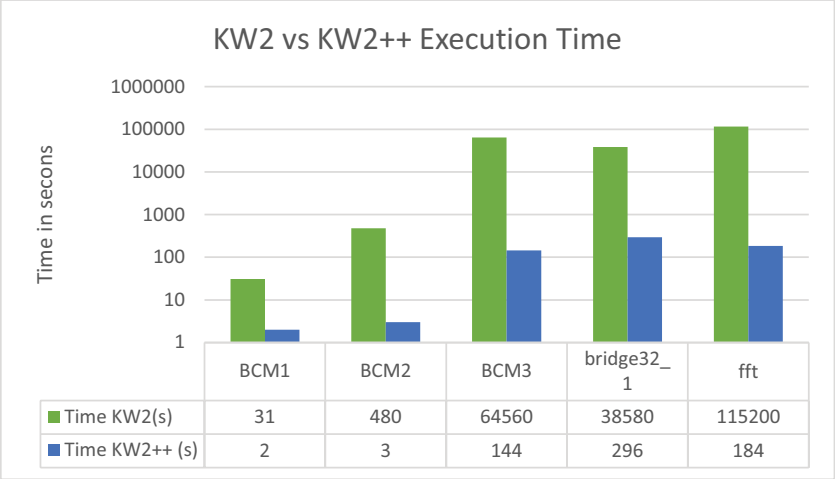


Figure 4.8: KW2 vs KW2++ execution time comparison

Last but not least, is is interesting focusing in the number of iterations needed. It is claimed that the original KW2 converges in an average of 25 iterations. After implementing this paper's KW2 algorithm, we noticed that the above claim would not stand, not only because the algorithm was spreading components too slow, but also because it was accepting every solution, regardless its correctness. So, in our KW2++ having this Upper Bound in the average movements of the components, we are rejecting invalid solutions and continue to the next placement iteration only if a right one is given. After experimentation, we present you the Figure's 4.9 results.

4.3 Focusing on our KW2++

Being really proud of KW2++ performance, we present you some resutls regarding execution time, wirelength, and iterations in the whole design set tested.

In table 4.4 we present you analytical results of all benchmarks. The are really remarkable as all goals set in the beginning are met. They are solved fast, they Converge fast, they preserve the best attainable wirelength and reach an semi-overlap free status. It should be noted that execution time is marked in minutes.

Figure 4.9: KW2 vs KW2++ iteration comparison

Table 4.4: KW2++ Main Results over 11 designs

Design Name	# Components	Iterations	Execution Time	Overlap	MaxDensity	WL
BCM1	716	9	0,03	0,184	4,089	36885,125
BCM2	2346	10	0,05	0,186	4,696	166518,255
BCM3	18796	13	2,5	0,186	5	1563070,04
bridge32_1	30675	30	5	0,193	5,257	2615598,998
fft	32281	23	3	0,191	5,145	8335183,294
cordic_I4	41601	29	2,75	0,197	5,225	3170817,087
des_perf_1	112644	31	31	0,189	5,972	10069376,808
matrix_mult	155325	25	21	0,197	6,912	43232911,791
b19	219268	20	37,75	0,187	17	45413475,128
leon3	649191	18	553	0,199	9	454233995,759
leon2	794286	20	1220	0,192	9,2	743909117,375

Figure 4.10 shows the number of iterations needed, for each benchmark to converge. Excluding the first 3 and over the last 8 benchmarks we concluded that an average number of 24 - 25 placement iterations are needed for our KW2++ to converge. It seams really fast and also experimentally proven.

An overall merging of the execution time for each design is also presented in Figure 4.11. Excluding the last two huge designs we have a really good execution time not longer than 40 minutes, which is an intriguing and interesting result and a need, for manufacturing companies. It should be noted that designs like leon3 and leon2 have a huge number of components. Usually, netlists of over 200.000 components are first clustered and then placed. That is the reason of my exclusion above. Nevertheless, KW2++ can handle 1mil component design really effeciently.

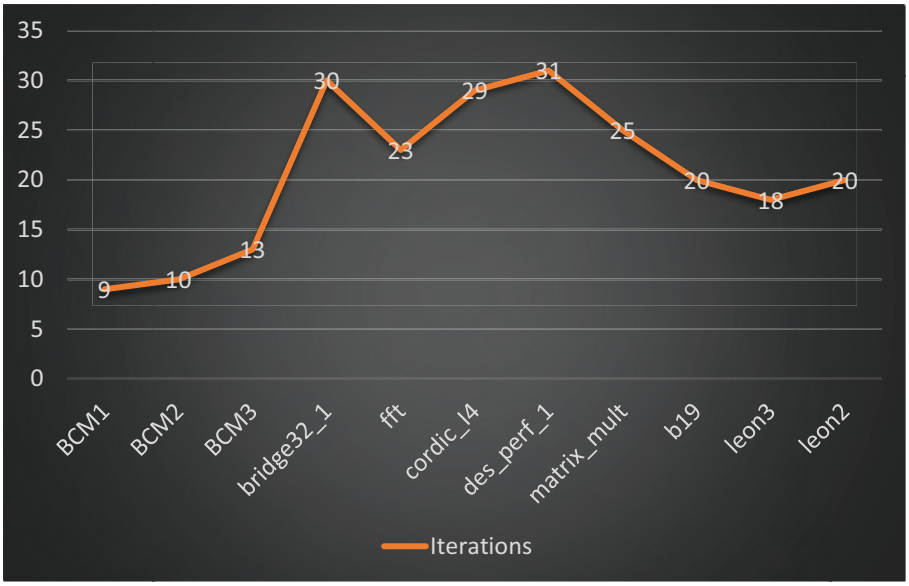
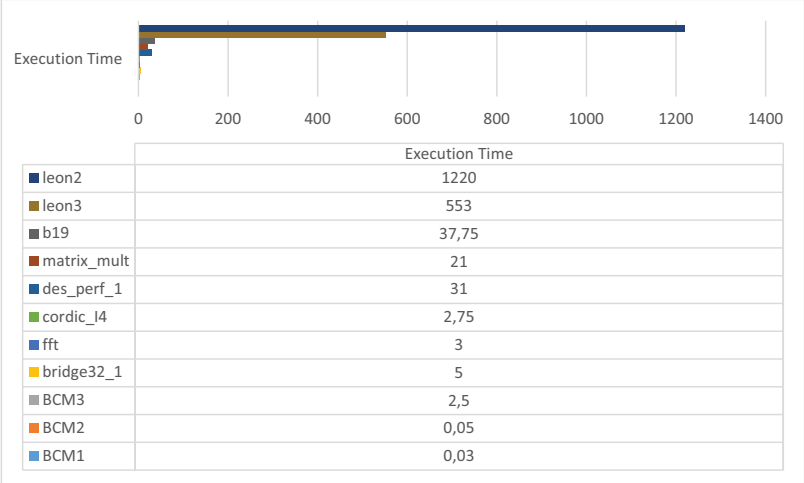


Figure 4.10: KW2++ number of iterations over the designs



	Execution Time
leon2	1220
leon3	553
b19	37,75
matrix_mult	21
des_perf_1	31
cordic_l4	2,75
fft	3
bridge32_1	5
BCM3	2,5
BCM2	0,05
BCM1	0,03

Figure 4.11: KW2++ execution time over the designs

CHAPTER 5

CONCLUSION AND FUTURE WORK

To sum up, Kraftwerk2++ achieved all our predefined goals. Its strength lies on the fact that it is proved fast in both Algorithmic Convergence and Execution Time. It is enriched with many features some of which are, a novel Quality Control system, Poisson and Gaussian Blur bin related formulation, Cluster support, different Aspect Ratio support and an integral Combinatorial Logic, which ensures the correctness of the ouput result. Supporting very large designs,

KW2++ can be easily extended and optimized in the future. It can be improved as follows:

- using a Pre-conditioner
- used in Timing Driven Placement
- used in 3D Placement
- optimize Clusters Support
- calibrate Gaussian Blurring Algorithm and possibly introduce a new one
- KW2++ execution in all Floorplan partitions simultaneously using Parallel methods
- combination with a Legalizer, like *Abacus2* to achieve iteratively better quality results

BIBLIOGRAPHY

[1] C. J. ALPERT, D. P. MEHTA, AND S. S. SAPATNEKAR, *Handbook of algorithms for physical design automation*, CRC press, 2008.

[2] G. COLLINS ET AL., *Fundamental numerical methods and data analysis*, Fundamental Numerical Methods and Data Analysis, by George Collins, II., (1990).

[3] A. B. KAHNG, J. LIENIG, I. L. MARKOV, AND J. HU, *VLSI physical design: from graph partitioning to timing closure*, Springer Science & Business Media, 2011.

[4] S. K. LIM, *Practical problems in VLSI physical design automation*, Springer Science & Business Media, 2008.

[5] G.-J. NAM AND J. J. CONG, *Modern circuit placement: best practices and results*, Springer Science & Business Media, 2007.

[6] N. A. SHERWANI, *Algorithms for VLSI physical design automation*, Springer Science & Business Media, 2012.

[7] P. SPINDLER, U. SCHLICHTMANN, AND F. M. JOHANNES, *Kraftwerk2—a fast force-directed quadratic placement approach using an accurate net model*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 27 (2008), pp. 1398–1411.

[8] T. F. E. WIKIPEDIA, *Placement (eda)*. [Online; accessed 16-September-2016].