MASTER THESIS

EXPLORATION OF TIMING-DRIVEN, ANALYTICAL GLOBAL PLACEMENT ALGORITHM

Student:
Dimitrios Gerostathis
dgerostathis@uth.gr

Supervisor:
Christos Sotiriou
chsotiriou@e-ce.uth.gr
Committee:
Georgios Stamoulis
georges@e-ce.uth.gr
Fotios Plessas
fplessas@e-ce.uth.gr

A thesis submitted in fulfillment of the requirements
for the degree of Master
in the
Circuits & Systems Laboratory (CASlab)
Department of Electrical and Computer Engineering

March 2, 2022

Abstract

In submicron VLSI circuits, interconnect delay dominates logic gate delay by a big factor. The former is proportional to its length which depends significantly on the placement step, where the locations of the logic and memory elements of the chip are determined. Originally the placement problem only satisfies layout constraints but recently modern placers also incorporate timing constraints and they are called timing-driven placers. In this master thesis, we present an approach for a Timing-Driven Global Placer which prioritizes the minimization of critical connections, by placing the corresponding components closer together. Criticality for a connection is determined by making use of slack information derived from Static Timing Analysis and comparing this value to the average of the slack distribution. The deviation from the average determines the amount of prioritization this connection gets and thus the amount of effort the placement algorithm gives, into placing the components corresponding to this connection, closer. Our approach shows promising results by reducing the total negative slack (TNS) and the worst negative slack (WNS) for a variety of designs relative to a conventional non timing-driven placement.

Περίληψη

Σε τεχνολογίες της τάξης των υπομικρόμετρων η καθυστέρηση της διασύνδεσης κυριαρχεί έναντι της καθυστέρησης πύλης. Η πρώτη είναι ανάλογη του μήκους καλωδίου της διασύνδεσης. Το τελευταίο εξαρτάται σε μεγάλο βαθμό απο το βήμα Τοποθέτησης στο οποίο η λογική και τα στοιχεία μνήμης τοποθετούνται στο τσίπ. Παραδοσιακά το πρόβλημα τοποθέτησης ικανοποιεί μόνο χωροταξικούς περιορισμόυς (πχ. εξουδετέρωση επικάληψης) αλλά τελευταία σύγχρονοι αλγόριθμοι τοποθέτησης λαμβάνουν υπόψιν τους και χρονικόυς περιορισμούς πέρνοντας το όνομα χρονικά-καθοδηγούμενοι τοποθετητές. Σε αυτήν την πτυχιακή παρουσιάζουμε μια προσέγγιση κατά την οποία δίνεται υψηλότερη προτεραιότητα σε κρίσιμες συνδέσεις με σκοπό να γίνουν μικρότερες μετακινόντας τα αντίστοιχα λογικά στοιχεία πιο κοντά μεταξύ τους. Η κρισιμότητα μιας σύνδεσης ορίζεται κάνοντας χρήση της slack πληροφορίας και συγκρινοντάς την με την μέση τιμή της συνολικής slack κατανομής. Το ποσο της προτεραιοτητας που δινεται σε μια συνδεση, εξαρταται απο το ποσο αποκλισης της, απο την μεση κατανομη. Η προσέγγιση μας δίνει καλά υποσχόμενα αποτελέσματα μειόνοντας το συνολικό αρνητικό slack και το χειρότερο αρνητικό slack για ένα έυρος δεσιγν, σε σχεση με εναν μη χρονικα καθοδηγουμενο τοποθετητη.

Acknowledgements

I would like to thank first and foremost my thesis supervisor Prof. Christos Sotiriou, who gave me the impetus to start and complete this study and who was always available and willing to help.

I would also like to thanks the CASlab members for being friendly and providing assistance and especially Christos Georgakidis, who helped me in my thesis research.

Finally, I would like to express my acknowledgements to my parents who supported me throughout my studies in University of Thessaly.

Dimitrios Gerostathis
Volos, 2022

Exploration of Timing-Driven, Analytical Global Placement Algorithm

Dimitrios Gerostathis
dgerostathis@uth.gr

Εξερεύνηση Αναλυτικού, Χρονικά-καθοδηγούμενου Αλγορίθμου Καθολικής Τοποθέτησης

Δημήτριος Γεροστάθης
dgerostathis@uth.gr

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

As integrated circuit technology advances, circuit performance becomes heavily dependent on interconnect wire delays. In submicron VLSI circuits, the wiring delay dominates the logic gate delay by a big factor. Wire delay is proportional to wire length and the latter depends significantly on the placement step which decides where the logic and memory elements of the chip are located.

In high-speed interconnect dominated designs, placement also has a significant impact on design functions such as buffering, gate resizing, logic synthesis and logic design which all help to meet timing requirements.

Originally the placement problem satisfies two main constraints which are the minimization of the wirelength between components and the neutralization of overlap. Recent modern placers also incorporate timing constraints with the aim of producing a layout of improved timing metrics, which will also help subsequent steps in the design flow, to reach timing closure easier.

In this integrated master thesis, we present an approach, to improve our already existed Global Placement algorithm, of our Electronic Design Automation (EDA) tool to be timing driven.

Our placement algorithm is analytic and force directed which means it models the minimization of the wirelength and neutralization of overlap as forces. The former is modeled as a spring force which increases with the increase of wirelength between two components. This force is called, net force. The latter is called move force and its purpose is to move components to overlap free areas. Both of these forces work in conjunction iteratively and converge to a final placement solution.

Every connection has a cost which represents the spring coefficient. Our method works by increasing the coefficients of critical connections to increase their net force. This is done by a net-weight assignment process which maps slack values to timing weights which are then multiplied to the spring coefficients.

Timing weights are generated from mapping functions and their value depends on the deviation of slack values from the average of the slack distribution. Slacks with values less than the average get increased weights and are considered critical, while slack with values greater than the average get reduced weights and are considered non-critical.

Our method shows promising results by substantially reducing the total negative slack (TNS) and the worst negative slack (WNS) for a variety of designs, relative to a conventional placement.

Chapter 2

Theoretical Background

2.1 Introduction to EDA

The integrated circuit revolutionized the electronics industry and paved the way for the development of all the world's technological marvels. It is without a doubt one of the most complex engineering products ever built. Since its invention, the number of transistors per integrated circuit has doubled every two years, following the now-famous Moore's Law. This exponential growth made ICs smaller and more powerful and transformed all areas of modern society with amazing inventions fitting in the palm of a hand.

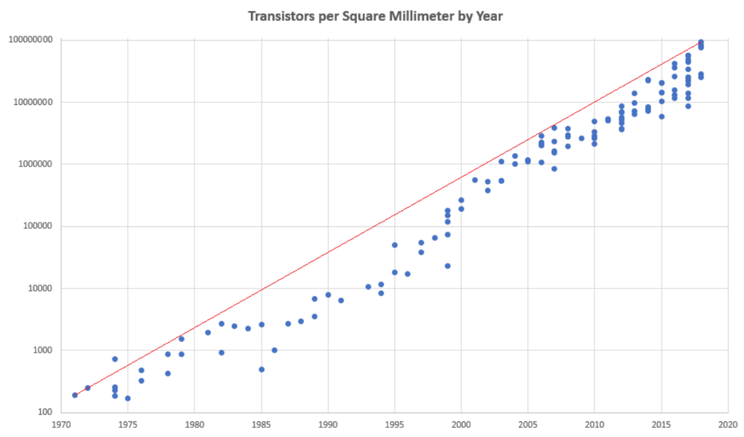


Figure 2.1: Moore's Law from 1970 till 2020

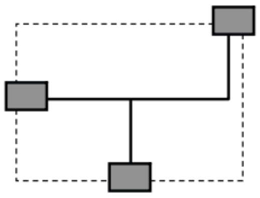
In the early years of semiconductor technology, the task of laying out gates and interconnect wires was carried out manually by hand. However as the semiconductor scaling began to drop this became from an increasing tedious task to a practically humanly impossible one. The use of automation to address the resulting problem of scale became of extreme importance. Automation was facilitated by the improvement in the speed of computers that would be used to create the next generation of computer chips resulting in their own replacement!

This gave birth to the development of Electronic Design Automation Software (EDA). The earliest and most fundamental problem these tools solved was physical design automation, that is the computation of the best physical layout of millions to billions of circuit components on a tiny silicon surface. However modern EDA tools do way more than that and they almost touch every aspect of the IC design flow, from high-level system design to fabrication. In today's nanometer process technologies interconnects play an ever increasing role in determining the overall circuit performance, power, area and cost. Also other phenomena like crosstalk and power grid noise began to rise as wire cross-sections became "taller and thinner" from one technology generation to the next. The physical arrangement of the components controls the quantity and quality of these interconnects directly and thus the circuit optimisation and pepformance.

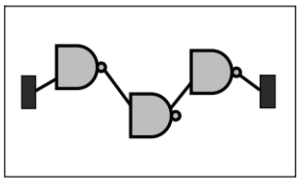
2.2 Placement

Placement is one of the most fundamental steps of the Physical Design flow. Its objective is to determine the locations and orientations of all circuit elements within a layout, given solution constraints and optimization goals. Placement goals are modeled and measured through the appropriate cost functions which need to be optimized. It is beneficial to use smooth cost functions which can be easily adopted in placement algorithms. The most common ones are: [1]

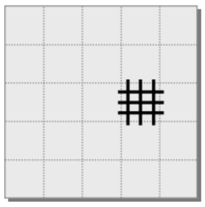
- *Total Wirelength:* This is the primary objective of most of the existing placers. That's because wirelength directly affects all of the other cost functions, especially in modern VLSI designs where wire delays are the dominating factor.



- *Delay:* Frequency of a chip is determined by the delay of its longest path known as the critical path. Given design constraints a placement algorithm must ensure that no such paths exist.



- *Congestion:* This is an important cost function for measuring routability which is the step that follows placement. 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,

Modern Placers are fairly complex and can combine different cost functions simultaneously. Also there is a strong interaction between them, for example delay depends on the total wirelength and power can depend on congestion. In figure Fig. 2.2 an example is shown between a bad placement and a good placement considering Total Wirelength as the cost function.

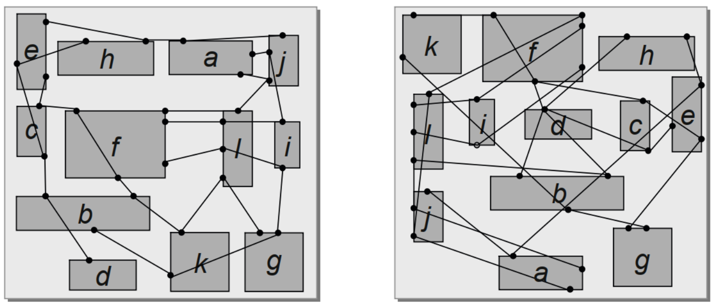


Figure 2.2: A comparison between a bad (left) and a good (right) placement, considering Total Wirelength.

The placement process is divided into three subsequent steps. These are the Global Placement, the Legalization and the Detailed Placement. In the following we explain each one of them.

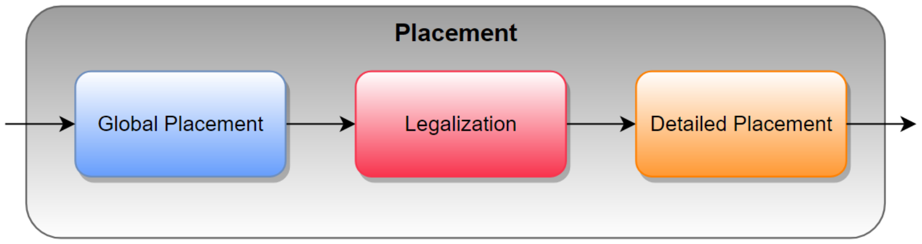


Figure 2.3: Placement Flow Steps

2.2.1 Global Placement

Global placement is the first step of the placement process. It focuses on finding an initial placement of the logic cells. Global placement often ignores cell specific shapes and sizes and does not attempt to align their location with valid gridrows and columns. It is mainly concerned with “rough” locations of the gates and allows an amount of overlap between them. This overlap is then removed in the Legalization step. Performance optimisations like timing improvement can also take place in this stage.

There are various techniques for circuit placement and can be summarized as follows. [1]

- Partitioning Based Algorithms: The netlist and the layout are divided into smaller sub-netlists and sub-regions, respectively, according to cut-based cost functions. This process is repeated until each sub-netlist and sub-region is small enough to be handled optimally. Each of these sub-regions must have the smallest dependence possible with other sub-regions i.e. small number of connections. An example of this approach is min-cut placement, which tries to minimize the "cut" of connections between partitions (sub-regions).
- Analytic Techniques: Model the placement problem using an objective cost function, which can be maximized or minimized via mathematical optimization methods. The objective can be quadratic or otherwise non-convex. The

most common objective is quadratic and aims at the minimization of the total wirelength. Quadratic forms used in global placement are convex and can me minimized by calculating their gradient to zero. This produces a series of linear equations and their solution define the module positions with the global minimum cost function value. A lot of placers model this gradient as a spring force and they are called, force-directed. Quadratic methods are relatively easy to implement and appear to be more scalable in terms of runtime, in large designs. The global placer described in this thesis is force-directed, based on an algorithm called Kraftwerk2.

- Stochastic Algorithms: Randomized moves are used to optimize the cost function. An example of this approach is simulated annealing. Simulated annealing is a metaheuristic to approximate global optimization in a large search space. The name of the algorithm comes from annealing in metallurgy. At each step, the simulated annealing heuristic considers some neighboring state s* of the current state s, and probabilistically decides between moving to the next state or not. The probability of movement is defined as temperature. In this algorithm the temperature progressively decreases from an initial positive value to zero to accept less and less worse solutions at each step. The acceptance of worse solutions is essential for the placement algorithm to be able to escape from local minimums of the cost function and aproximate the global minimum solution.

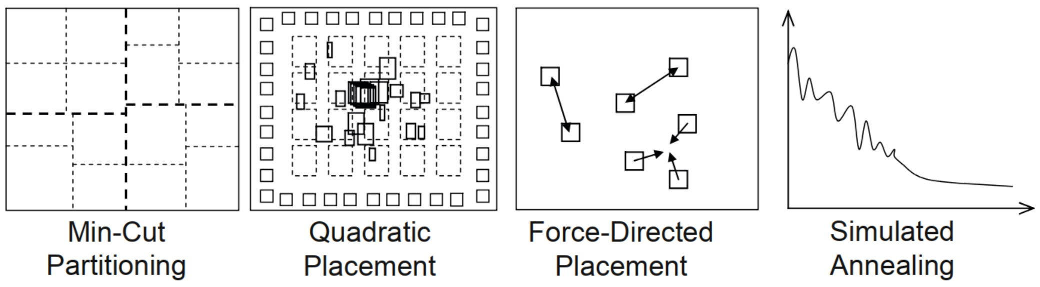


Figure 2.4: Most common Placement tecniques.

2.2.2 Legalization

Legalization seeks to move cells in their legal positions, that is positions with no overlap and with total alignment with the power grid, the rows and the columns of the chip layout. It tries to achieve that while simultaneously seeking to minimise displacement from global placement to have the minimum impact possible on wirelength and circuit delay. It's efficiency strongly depends on the quality of the global placement result. Legalization is not necessary only after global placement but also after incremental changes, such as resizing and buffering.

2.2.3 Detailed Placement

Detailed Placement aims to improve the solution further concerning given objectives. It incrementally improves the location of each cell by local operations, such as swapping neighbouring cells to reduce total wire length, or shifting several cells in a row to create room for another object when white space is available.

2.3 Global Placement

In this chapter we will give an extensive overview regarding the mathematical formulation of the global placement problem. The purpose of this thesis is the development of a Timing Driven Global Placer so the knowledge described, is fundamental for the understanding of subsequent chapters.

2.3.1 Analytic Problem Formulation

Analytic placement minimizes a given objective, such as wirelength or circuit delay, using mathematical optimization techniques. The given objective takes the form of a cost function and must be minimized. Such techniques often require certain assumptions, such as the differentiability of the objective function or the treatment of placeable cells as dimensionless points. The most common cost function used is quadratic. That is because quadratic minimization facilitates the calculation of partial derivatives (gradient), rather than using a linear objective. Another benefit is that it emphasizes more on the minimization of longer connections. The general form of a quadratic cost function is given below. It's goal is to minimize the squared euclidean distance between the components [2].

$$\Gamma(\mathbf{x},\mathbf{y}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} c(i,j)\Big((x_i - x_j)^2 + (y_i - y_j)^2\Big) \tag{2.1}$$

Where $\mathbf{x} = (x_1, x_2, ..., x_n)^T$, $\mathbf{y} = (y_1, y_2, ..., y_n)^T$ are the cell positions in the horizontal (x) and vertical (y) direction respectively, n is the total number of cells and *c(i,j)* is the connection cost between cells i and j. If two cells are unconnected then *c(i,j)* = 0. Terms $(x_i - x_j)^2$ and $(y_i - y_j)^2$ correspond to the squared horizontal and vertical euclidean distances between cells i and j. Components are considered dimensionless and $\mathbf{x}$, and $\mathbf{y}$ vectors, normally represent their center.

The term *c(i,j)* is very important for the timing driven placers and it is explained in the following chapter. Timing driven global placers multiply the connection cost *c(i,j)*, by some weights that they generate based on certain design metrics, to increase the priority of critical nets. For example nets with negative slack, get increased connection costs to guide the minimization objective into keeping them shorter.

2.3.2 Vector Notation and Connectivity Matrix

Next we will show how the cost function Γ (2.1) can be represented in a vector-matrix form notation [3]. This is an essential step because this form, utilizes the solving capabilities of computer software, specific on solving multi variable linear equations, represented by high order sparse matrices. The cost function Γ (2.1) can be separated in the x and y direction and each part can be minimized independently. It is important to note that x_i and x_j must be different at all times because two components cannot point to the same place. Also this constraint ensures that the trivial solution of $\mathbf{x} = \vec{0}$ is avoided.
By working on the x direction we obtain the following:

$$
\begin{aligned}
\Gamma_x(\mathbf{x}) &= \frac{1}{2}\sum_i\sum_j c_{ij}(x_i - x_j)^2 && \text{(2.2a)}\\
&= \frac{1}{2}\sum_i\sum_j c_{ij}(x_i^2 - 2x_ix_j + x_j^2) && \text{(2.2b)}\\
&= \frac{1}{2}\Big(\sum_i c_{ii}x_i^2 - 2\sum_i\sum_{j\neq i} c_{ij}x_ix_j + \sum_j c_{jj}x_j^2\Big) && \text{(2.2c)}\\
&= \frac{1}{2}\Big(\sum_i c_{ii}x_i^2 - 2\sum_i\sum_{j\neq i} c_{ij}x_ix_j + \sum_i c_{ii}x_i^2\Big) && \text{(2.2d)}\\
&= \frac{1}{2}\Big(2\sum_i c_ix_i^2 - 2\sum_i\sum_{j\neq i} c_{ij}x_ix_j\Big) && \text{(2.2e)}\\
&= \sum_i c_ix_i^2 - \sum_i\sum_{j\neq i} c_{ij}x_ix_j && \text{(2.2f)}\\
&= \mathbf{x}^{\mathbf{T}}\mathbf{D}\mathbf{x} - \mathbf{x}^{\mathbf{T}}\mathbf{C}\mathbf{x} && \text{(2.2g)}\\
&= \mathbf{x}^{\mathbf{T}}(\mathbf{D} - \mathbf{C})\mathbf{x} && \text{(2.2h)}\\
\Gamma_x(\mathbf{x}) &= \mathbf{x}^{\mathbf{T}}\mathbf{L}\mathbf{x} && \text{(2.2i)}\\
& && \text{(2.2j)}
\end{aligned}
$$

The above transformation is based on the following equality:
Given any quadratic form with n variables, its coefficients can be arranged into a n $\times$ n symmetric matrix A, where:

$$\sum_i\sum_j a_{ij}x_ix_j = \mathbf{x}^{\mathbf{T}}\mathbf{A}\mathbf{x}$$

Using the former, we can easily jump from equation (2.2e) to equation (2.2f), where $\mathbf{C}$ and $\mathbf{D}$ are called the connection matrix and diagonal matrix respectively. Also for an ease of notation we represent c_{ii} as c_i and c_{jj} as c_j.

The connection matrix $\mathbf{C}$, is a symmetric matrix that stores information about every connection between the components of the design. More specifically between two components i, j it stores their connection costs c_{ij} and c_{ji}, at matrix positions $\mathbf{C}(i,j)$ and $\mathbf{C}(j,i)$ and has zero in the diagonal positions.

The diagonal matrix $\mathbf{D}$, at entry $\mathbf{D}(i,i)$ stores the summation of every connection cost between component i and the components it connects to, i.e. every connection cost c_{ij} updates both the $\mathbf{D}(i,i)$ and $\mathbf{D}(j,j)$ entries.

The laplacian matrix $\mathbf{L}$, is formed by subtracting the connectivity matrix by the diagonal matrix and constructs the final form of the vector-matrix notation of the quadratic cost function.

In equation (2.2b) and below, the constraint $i \neq j$ is specifically used.
At equation (2.2d), $\sum_i c_{ii}x_i^2 = \sum_i c_{jj}x_j^2$ because although $x_i \neq x_j$ their summations over the whole domain, are.

In a real chip, not all points are movable. Ports (I/Os) and some specific cells (e.g memory elements) stay fixed throughout the placement procedure. If we also take these points into consideration then the cost function transforms into the equation (2.3) which has the addition of a linear and a constant term. The connection between a movable cell x_i and a fixed point x_f is evaluated as:

$$c_{if}(x_i - x_f)^2 = c_{if}x_i^2 - 2c_{if}x_ix_f + c_{if}x_f^2 = c_ix_i^2 - 2c_{if}x_ix_f + c_fx_f^2$$

The first term contributes to the diagonal matrix D, by adding on row i all the connection coefficients between the movable node i and the fixed point f, c_{if}. The second linear term contributes to the vector $\mathbf{b}^T$, by adding on its row i all the connection coefficients between the movable node i and the fixed point f, c_{if} multiplied by the fixed point x-position x_f. That is for every line of the $\mathbf{b^T}$ vector, $b_i = -\sum_f c_{if}x_f$. The third term is constant and contributes to the *const* part.

$$\Gamma_x(\mathbf{x}) = \mathbf{x^T L x} + \mathbf{b^T x} + const \tag{2.3}$$

The coefficients c_{if} are normally stored into a pin connection matrix P, which keeps information on every connection between the movable and fixed points.

Now we will show an example for a sample circuit, and derive the aforementioned matrices. The circuit in figure 2.5 consists of four inputs $i1, i2, i3, i4$, seven logic gates and four outputs $o1, o2, o3, o4$.

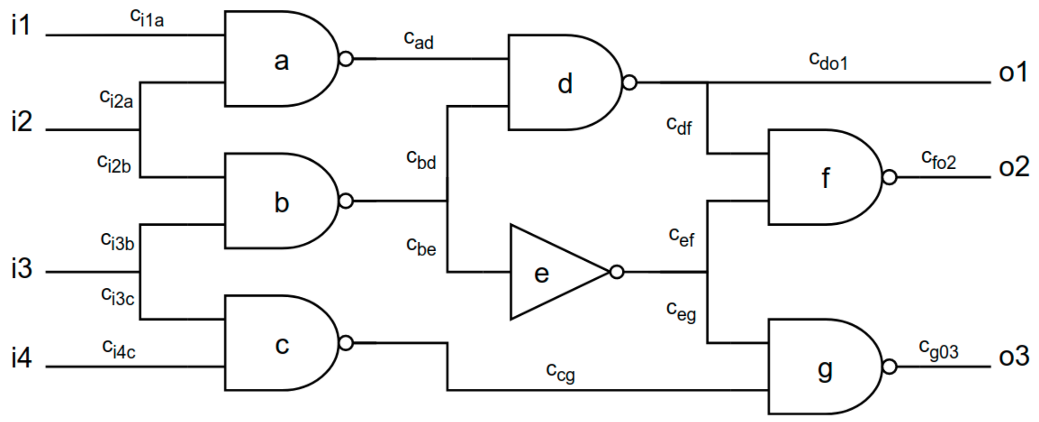


Figure 2.5: Sample Circuit

There are seven gates in total so the connectivity matrix is 7x7. It stores every connection between the movable gates. Inputs and outputs are considered fixed points.

$$\mathbf{C} = \begin{bmatrix} 0 & 0 & 0 & c_{ad} & 0 & 0 & 0 \\ 0 & 0 & 0 & c_{bd} & c_{be} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & c_{cg} \\ c_{da} & c_{db} & 0 & 0 & 0 & c_{df} & 0 \\ 0 & c_{eb} & 0 & 0 & 0 & c_{ef} & c_{eg} \\ 0 & 0 & 0 & c_{fd} & c_{fe} & 0 & 0 \\ 0 & 0 & c_{gc} & 0 & c_{ge} & 0 & 0 \end{bmatrix}$$

The diagonal matrix for a cell at point i, stores the summation of its connections between both the movable cells and the fixed points. For example gate a connects with gate d and with the inputs $i1$, $i2$.

$$\mathbf{D} = \begin{bmatrix} d_a & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & d_b & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & d_c & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & d_d & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & d_e & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & d_f & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & d_g \end{bmatrix}$$

$$\begin{aligned} d_a &= c_{i1a} + c_{i2a} + c_{ad} \\ d_b &= c_{i2b} + c_{i3b} + c_{bd} + c_{be} \\ d_c &= c_{i3c} + c_{i4c} + c_{cg} \\ d_d &= c_{da} + c_{db} + c_{do1} + c_{df} \\ d_e &= c_{da} + c_{db} + c_{do1} + c_{df} \\ d_f &= c_{fd} + c_{fe} + c_{fo2} + c_{be} + c_{fo1} \\ d_g &= c_{ge} + c_{gc} + c_{go3} \end{aligned}$$

The Laplacian matrix is the subtraction between the diagonal matrix minus the connection matrix.

$$\mathbf{L} = \begin{bmatrix} d_a & 0 & 0 & -c_{ad} & 0 & 0 & 0 \\ 0 & d_b & 0 & -c_{bd} & -c_{be} & 0 & 0 \\ 0 & 0 & d_c & 0 & 0 & 0 & -c_{cg} \\ -c_{da} & -c_{db} & 0 & d_d & 0 & -c_{df} & 0 \\ 0 & -c_{eb} & 0 & 0 & d_e & -c_{ef} & -c_{eg} \\ 0 & 0 & 0 & -c_{fd} & -c_{fe} & d_f & 0 \\ 0 & 0 & -c_{gc} & 0 & -c_{ge} & 0 & d_g \end{bmatrix}$$

Finally the **b** vector is defined using $b_i = -\sum_f c_{if} x_f$ as:

$$\mathbf{b} = \begin{bmatrix} b_a \\ b_b \\ b_c \\ b_d \\ b_e \\ b_f \\ b_g \end{bmatrix} = \begin{bmatrix} -\sum_f c_{af} x_f \\ -\sum_f c_{bf} x_f \\ -\sum_f c_{cf} x_f \\ -\sum_f c_{df} x_f \\ -\sum_f c_{ef} x_f \\ -\sum_f c_{ff} x_f \\ -\sum_f c_{gf} x_f \end{bmatrix} = \begin{bmatrix} c_{ai1} x_{i1} + c_{ai2} x_{i2} \\ c_{bi2} x_{i2} + c_{bi3} x_{i3} \\ c_{ci3} x_{i3} + c_{ci4} x_{i4} \\ c_{do1} x_{o1} \\ 0 \\ c_{fo2} x_{o2} \\ c_{go3} x_{o3} \end{bmatrix}$$

2.3.3 Cost function minimization

The (2.3) cost function can be minimized by using mathematical optimisation techniques. The laplacian matrix L is symmetric and for the majority of the designs, it has high sparsity. If no fixed points exist, L is positive semi-definite, i.e.

- $\mathbf{x^T} L \mathbf{x} \geq 0$ for every vector $\mathbf{x} = (x_1, ..., x_n)^T$

and if fixed modules exist then the laplacian matrix is positive definite, i.e.

- $\mathbf{x^T} L \mathbf{x} > 0$ for every vector $\mathbf{x} = (x_1, ..., x_n)^T$

A quadratic form is definite if it is zero, only when all of its variables are zero simultaneously.
Due to the above the The (2.3) cost function is convex, and its minimum is obtained by calculating the points which set its gradient to zero. In vector calculus, the gradient of a scalar field f in the space $\mathbf{R}^n$ is the transpose of the derivative of a scalar by a vector.

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x1} \\ \vdots \\ \frac{\partial f}{\partial xn} \end{bmatrix} = \left(\frac{\partial f}{\partial \mathbf{x}} \right)^T$$

So minimum will be obtained by solving the following equation:

$$\nabla\Gamma_x = \vec{0} \Rightarrow \left(\frac{\partial\Gamma_x}{\partial\mathbf{x}}\right)^T = \vec{0} \tag{2.4}$$

The first step for deriving the gradient of the cost function, is to calculate its partial derivative. For this purpose, we make use of the following identity on vector-by-vector partial derivatives, obtained by the matrix calculus math theory.

$$\frac{\partial(\mathbf{u}\cdot\mathbf{A}\mathbf{v})}{\partial\mathbf{x}} = \frac{\partial\mathbf{u}^\mathbf{T}\cdot\mathbf{A}\mathbf{v}}{\partial\mathbf{x}} = \mathbf{u}^\mathbf{T}\mathbf{A}\frac{\partial\mathbf{v}}{\partial\mathbf{x}} + \mathbf{v}^\mathbf{T}\mathbf{A}^\mathbf{T}\frac{\partial\mathbf{u}}{\partial\mathbf{x}}$$

Based on the former, we calculate the partial derivative of the quadratic term of the cost function by setting $\mathbf{u} = \mathbf{v} = \mathbf{x}$ and $\mathbf{A} = \mathbf{L}$.

$$\begin{aligned}
\frac{\partial(\mathbf{x}^\mathbf{T}\cdot\mathbf{L}\mathbf{x})}{\partial\mathbf{x}} &= \mathbf{x}^\mathbf{T}\mathbf{L}\frac{\partial\mathbf{x}}{\partial\mathbf{x}} + \mathbf{x}^\mathbf{T}\mathbf{L}^\mathbf{T}\frac{\partial\mathbf{x}}{\partial\mathbf{x}} && \text{(2.5a)}\\
&= \mathbf{x}^\mathbf{T}\mathbf{L}\mathbf{I} + \mathbf{x}^\mathbf{T}\mathbf{L}^\mathbf{T}\mathbf{I} && \text{(2.5b)}\\
&= \mathbf{x}^\mathbf{T}\mathbf{L} + \mathbf{x}^\mathbf{T}\mathbf{L}^\mathbf{T} && \text{(2.5c)}\\
&= \mathbf{x}^\mathbf{T}(\mathbf{L} + \mathbf{L}^\mathbf{T}) && \text{(2.5d)}
\end{aligned}$$

Finally we proceed to the calculation of the gradient:

$$\begin{aligned}
\nabla\Gamma_x &= \nabla(\mathbf{x}^\mathbf{T}\cdot\mathbf{L}\mathbf{x} + \mathbf{b}^\mathbf{T}\mathbf{x} + \mathit{const}) && \text{(2.6a)}\\
&= \left(\mathbf{x}^\mathbf{T}(\mathbf{L} + \mathbf{L}^\mathbf{T}) + \mathbf{b}^T + 0\right)^T && \text{(2.6b)}\\
&= (\mathbf{L} + \mathbf{L}^T)\mathbf{x} + \mathbf{b} && \text{(2.6c)}\\
&= 2\mathbf{L}\mathbf{x} + \mathbf{b} && \text{(2.6d)}
\end{aligned}$$

In the calculations of 2.6a and 2.5d we make use of the matrix identities, $(\mathbf{A} + \mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$ and $(\mathbf{A}\mathbf{B})^T = \mathbf{B}^T\mathbf{A}^T$. Additionaly because as aforementioned,the laplacian matrix $\mathbf{L}$ is symmetric, $\mathbf{L} = \mathbf{L}^T$. That's why eq: 2.6d follows eq: 2.6c.

Finally equation (2.4) becomes:

$$\begin{aligned}
\nabla\Gamma_x &= 2\mathbf{L}\mathbf{x} + \mathbf{b} = \vec{0} && \text{(2.7a)}\\
\Rightarrow 2\mathbf{L}\mathbf{x} &= -\mathbf{b} && \text{(2.7b)}
\end{aligned}$$

From 2.7b we obtain a n x n system of linear equations which can be easily solved by linear solvers like the intel MKL.

Often in the literature the cost function 2.3 is re-written as:

$$\Gamma_x(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\mathbf{T}\mathbf{L}\mathbf{x} + \mathbf{b}^\mathbf{T}\mathbf{x} + \mathit{const}$$

which is the general form of the quadratic cost function to eliminate the two in 2.7b and produce:

$$\mathbf{L}\mathbf{x} = -\mathbf{b}$$

2.3.4 Net Modeling

After the completion of the placement stage, the routing stage follows. During the routing stage, all the different component connections are routed using horizontal and vertical metal wires. One fundamental question that arises in the placement stage, is how the wirelength is going to be measured and minimized, when in reality wire does not exist. One solution would be to route the design after each placement iteration and measure the real wirelength, but that is totally futile, due to the enormous CPU time to complete the task. For this purpose it is essential to use wire estimations before routing and with that, drive the placement stage accordingly.

Those estimations are called net models. Before net models are described, first we are going to describe, what is a net in the first place. A net is simply a connection between components or more specifically a connection between an output pin of a component and the input pins of the components it connects to. During the placement stage, it is known from the design netlist which components connect to which i.e. the connectivity information of the circuit.

In fig 2.6 a circuit is shown with connections between three AND gates and two OR gates. Yellow dots are the output pins of the gates driving the nets. Blue dots are the input pins of the sinks i.e. the components connecting to the drivers. Net *a* and net *c* are two-pin nets and net *c* is a three-pin net. A net can connect with an indefinite number of pins, however modern designs consist of two and three-pin connections at more than 70%.

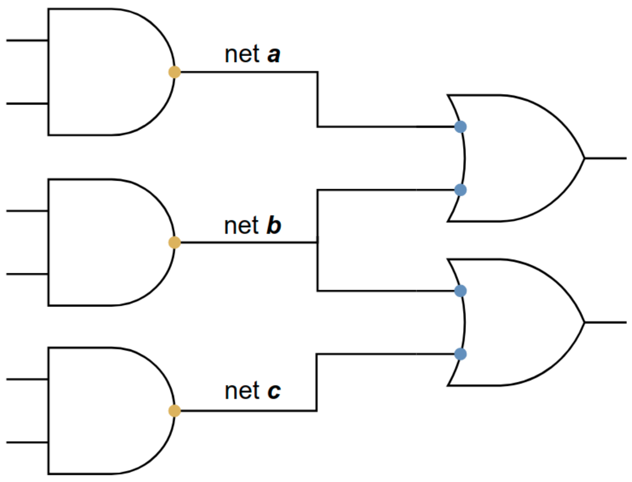


Figure 2.6: Net Definition

Another term used in the literature is called edge. Edge is denoted as a connection between two pins. For example net *b* contains two edges which are the connections between the upper and lower OR gate and nets *a*, *c* contain one edge each. Cost functions, like (2.1) attempt to minimize the euclidean distance of the edges of every net.

To summarize, every output pin of a component defines a net, every net consists of edges which are two-pin connections, and edges are minimized by the cost function.

The circuit netlist is represented as an undirected hyper-graph $G = (\mathcal{V}, \mathcal{N})$, where $\mathcal{V}$ are the vertices i.e. the circuit elements of the design and $\mathcal{N}$ are hyper-edges i.e. nets that connect them. The vertex set $\mathcal{N}$ consists of two disjoint subsets, MV and FV which are the set of movable and fixed circuit elements respectively.

Each net $n_i \in \mathcal{N}$ is represented as a hyper-edge which defines a subset of circuit elements, that are electrically connected to each other, i.e.
$n_i = \{\mathrm{v}_{i1}, \mathrm{v}_{i2}, \ldots, \mathrm{v}_{im}\}, \forall\, \mathrm{v}_{ij} \in \mathcal{V}$. Fig 2.7 shows the hyper-graph definition. Hyper-edges include multiple vertices (cells) and represent nets that are shown with different colors. The number of pins of the net (same as number of cells), is denoted by its cardinality $|n|$.

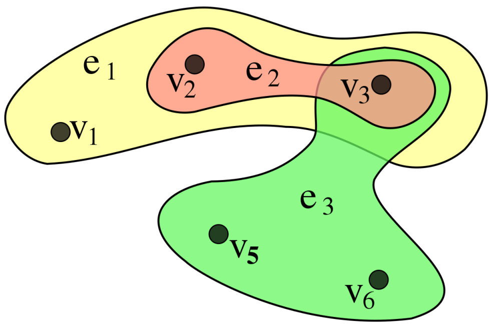


Figure 2.7: Hyper-Graph Definition

Some class of global placement algorithms, such as partitioning-based or simulated annealing, are effective in directly handling hyper-edge nets. However analytical placement algorithms, require the transformation of the hyper-edge to a set of two pin edges.

In the case of quadratic placement, each net $n \in \mathcal{N}$, is represented by a set of two-pin edges $e \in \mathcal{E}_n$, which connect the pins of the net. One edge is defined as $e = (p, q)$ and connects pins p and q. During quadratic placement, components are considered dimensionless and their positions in space can be represented by a single point (pin) which lies in their geometric center. Then the quadratic form 2.1 can be re-written as:

$$\Gamma(\mathbf{x}, \mathbf{y}) = \frac{1}{2} \sum_{n \in \mathcal{N}} \sum_{e \in \mathcal{E}_n} c(p, q)\Big((x_p - x_q)^2 + (y_p - y_q)^2\Big) \tag{2.8}$$

Net models simply specify the set $\mathcal{E}_n$, i.e. which edges represent the net $\mathcal{N}$. Nets are extracted based on the netlist, which specifies the connectivity information of the design. Net models are essential first and foremost for the approximation of the real wirelength of the design, i.e. the wirelength after the completion of the routing stage and secondly for the creation of symmetric and sparse Laplacian matrices which lead to the proper minimization of the objective function.

Theoretically the best net model would be the direct calculation of the wirelength after finalized routing. However this is totally futile, because placement is done iteratively, and the addition of the CPU time to route on each iteration would be enormous.

Considering the approximation of the routed wirelength, the most commonly used net model is the Half Perimeter Wire Length model (HPWL) because it is computationally fast and efficient. It is very accurate for nets with cardinality of two and three which constitute more than 80% of the total nets of the design. In fig 2.8 the HPWL definition is given. A net is defined between the output gatepin (yellow dot) of the gate on the left and the input gatepins (blue dots) of the gates it connects to. The dotted line corresponds to edges of the net which electrically connect the components. For the calculation of the HPWL, a bounding box is drawn which is the smallest rectangle enclosing all the pins of the net. As the name implies the HPWL is equal to the half perimeter of the bounding box, i.e. $HPWL = x_{max} - x_{min} + y_{max} - y_{min}$.

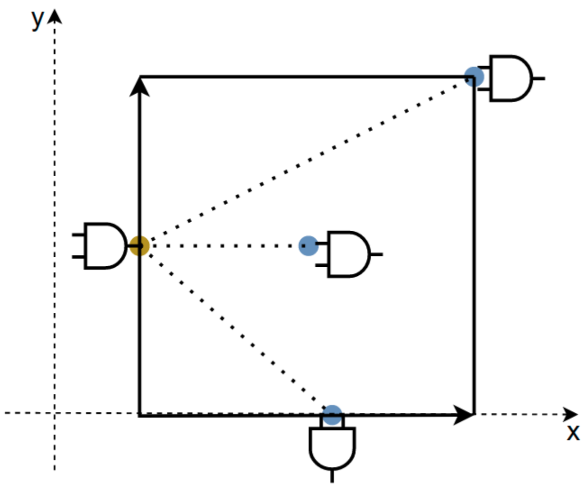


Figure 2.8: HPWL net model

Based on the previous, equation 2.8, becomes:

$$\Gamma(\mathbf{x}, \mathbf{y}) = \frac{1}{2} \sum_{n \in \mathcal{N}} (x_n^{max} - x_n^{min}) + (y_n^{max} - y_n^{min}) \tag{2.9}$$

Although HPWL is a very good estimation of the routed wirelength, it can not be used explicitly in the cost function, because it is not quadratic and cannot be represented in the matrix-vector notation form of 2.2, which leads to the inability of following the previous matematical analytical minimization techniques.

The most straightforward net model, which agrees with the quadratic mathematical formulation, is called point to point (P2P). This is derived directly based on the connectivity information of the net. Figure 2.9 shows a net with three edges (dotted lines) included in the $\mathcal{E}_n$ set. The euclidean distances which are calculated in the cost function are represented as solid arrowed lines.

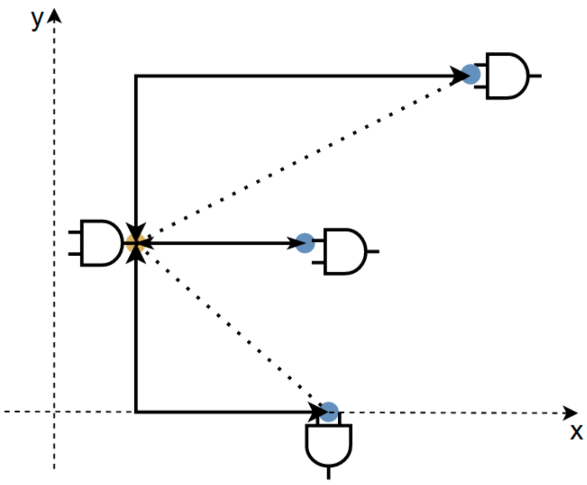


Figure 2.9: P2P net model

The extension of P2P, is the clique net model which includes all the possible edges of a net to create the $\mathcal{E}_n$ set. Both P2P and clique net models work well with quadratic placers because they create symmetric positive definite linear system equations which aid the solving efficiency of the global placer solvers. The advantage of the clique net model is the fact that it creates more dense matrices than the P2P model due to the increased number of edges and facilitates more on the minimization of the cost function. Figure 2.10 shows the clique net model. Red lines are the clique edges. These edges do not exist in the design netlist and they are simply an invention to keep the cells of a net, closer together. If the clique model is used the cost function 2.1 takes the form of:

$$\Gamma(\mathbf{x},\mathbf{y}) = \frac{1}{2}\sum_{p=1}^{|n|}\sum_{q=p+1}^{|n|} c(p,q)\Big((x_p - x_q)^2 + (y_p - y_q)^2\Big)$$

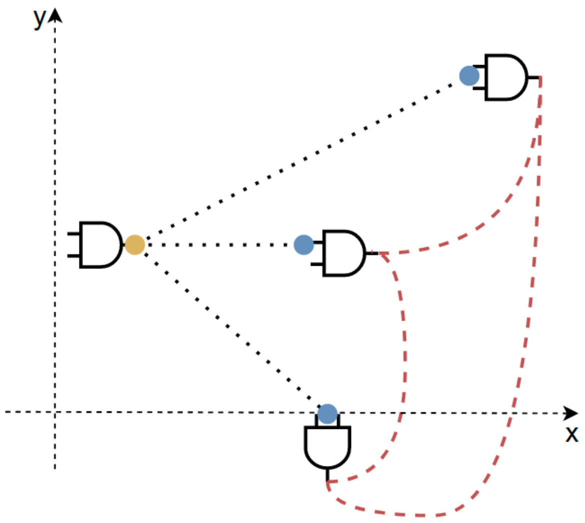


Figure 2.10: Clique net model represented with component dimensions

Because the clique net model creates complete graphs with far more edges than necessary to connect the net, it creates dense matrices. The latter, hinders the solving efficiency of the matrix solvers. A model that works well with large nets ($|n| > 5$) is the star net model shown in fig 2.11. In this, an extra pin called star is created, located at the center of mass of the net. Then every pin of the net is connected to the star pin. In the cost function calculation, only the star edges (red lines) are included. Star pins are useful because they entail only a linear number of graph edges.

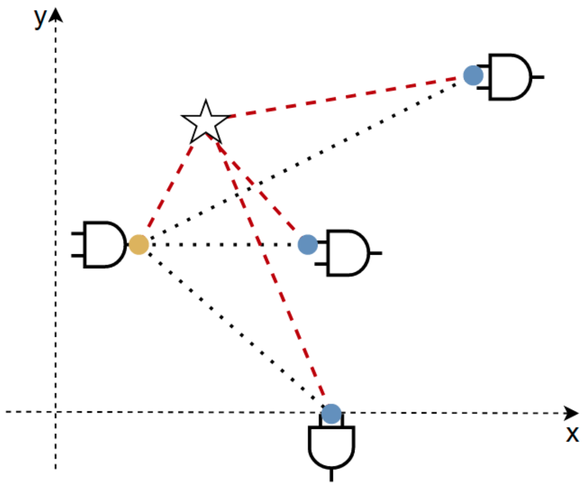


Figure 2.11: Star net model

Some placers combine different net models together in the so called hybrid models. For example the clique net model works well for small nets and star net model for bigger nets. By using both of them, maximum efficiency is achieved. Also clique and star net models can become equivalent if proper connection costs are defined. For example if c_{ij} is scaled with $1/|n|$, where $|n|$ is the cardinality of the net, then the clique net model is equivalent to the star net model [4].

Other more complex net models also exist, for special nets with high fanout like a clock tree. A minimum spanning tree (MST) net model, calculates a minimum tree length, which connects all pins of a net. A steiner tree (ST) net model, works similar to MST but it also makes use of arbitrary points used to branch off other tree segments and reduce the length even more. These models optimize for better routability for some difficult nets but are rarely used because of the scarcity of their occurrence.

2.4 Timing

Satisfying geometric requirements such as non-overlapping cells and routability, is not enough to produce a working integrated circuit. Meeting timing constraints is essential and the optimization process to achieve that is called timing closure. Timing closure uses specific methods like transistor resizing and buffering that alters the physical synthesis of the chip. However in modern EDA tools timing closure is integrated in every aspect of the design flow like Timing-Driven Placement which is the main topic of this thesis. Since now days wire delay is the main contributor to circuit delay, placement has a huge impact on wire length, routability and thus timing closure.

2.4.1 Propagation Delay

Ideally a logic gate responds immediately to a change at its input. However in the real world that's of course not the case due to the various parasitic capacitances seen at the output of the gate. The operational speed of a digital system is determined by the propagation delay of the logic gates used for its construction. The inverter is the fundamental logic gate of any integrated circuit and for that, its propagation delay is crucial for determining the speed of a given technology.

The Complementary Metal Oxide Semiconductor (CMOS) inverter consists of a PMOS and NMOS transistors which act as voltage controlled switches which connect the output to either VDD or GND. Figure 2.12 depicts a CMOS inverter with a parasitic capacitance Cload, seen at the output pin. This capacitance consists of the output parasitic capacitance of the inverter itself, the interconnect capacitance of the net the output pin defines and the input capacitance of the gates it connects to. The speed of the inverter is determined by how fast the capacitor is charged to VDD or discharged to GND [5].

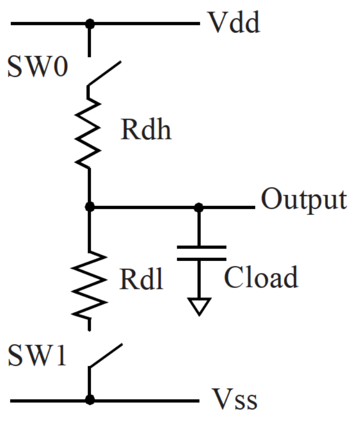


Figure 2.12: CMOS inverter structure

Propagation delay is normally measured as the time it takes for the output pulse to pass through the half-point of its excursion. Because the input pulse is not ideal itself, the measurement is made as the difference between the 50% point of the input pulse and the 50% point of the output pulse. There is also a distinction between two different propagation delays, the fall delay and the rise delay, corresponding to the output pulse. Propagation delay definition is illustrated figure 2.13.

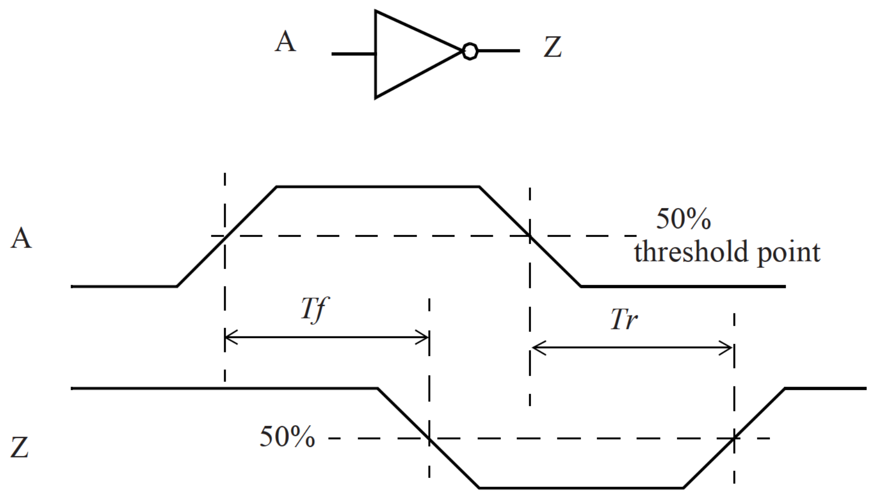


Figure 2.13: Propagation delay definition

2.4.2 Synchronous Digital Circuit

A synchronous integrated circuit is structured based on two distinct entities. The sequential logic and the combinational logic. The former consists of storage elements which store the bit information and change state based on the clock cycles. For example flip flops change state at every positive edge of the clock.

At every clock period, bits move from one storage element to other through the combinational logic. These two entities combined implement the working synchronous principle of the integrated circuit.

The two points of interest are the output of a launching memory element and the input of capture memory element separated by the combinational logic. In a circuit combinational logic is what induces the propagation delay on a signal. Figure 2.14 depicts what's described so far. A flip flop launches data at the positive edge of the clock and at the next clock edge the other flip flop captures the data.

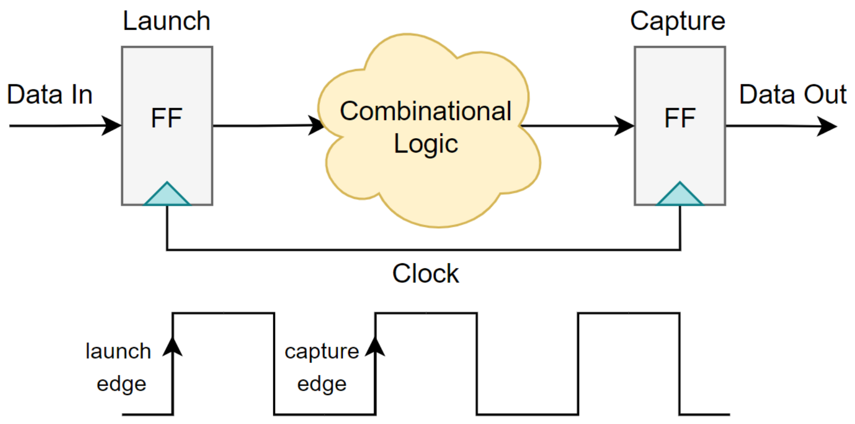


Figure 2.14: Synchronous digital circuit

2.4.3 Timing Constraints

To make sure data is launched and captured properly in a synchronous digital circuit, certain timing constraints must be met. These are:

- *Setup constraint:* Specifies the amount of time a data input should be stable before the clock edge of a storage element (flip flop or latch).
- *Hold-time constraint:* Specifies the amount of time a data input signal should stay stable after the clock edge at a storage element (flip flop or latch).

In Fig. 2.15 a representation of these constraints can be seen. With t_{su} and t_{hd} we represent the setup and hold constraints respectively, C is the clock and D is the input data into the storage element.

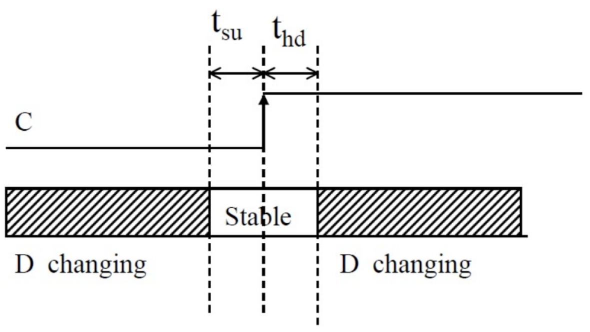


Figure 2.15: Setup and hold violations

Setup constraints ensure data launched from a launch flip flop at the the previous clock cycle, arrives *early* enough to be captured by a capture flip flop at the current clock cycle. Because the signal needs to arrive fast, setup constraints are checked on the longest paths of the design i.e. paths with biggest delay.

Hold constraints ensure that data launched by the current clock cycle arrive *late* enough so they do not overwrite the data at the same current clock cycle of the capture flip flop i.e. the capture flip flop has enough time to propagate the value of the previous clock cycle. Because the signal needs to arrive late, hold constraints are checked on the shortest paths of the design i.e. paths with smallest delay.

2.4.4 Timing Closure

The process of satisfying the above timing constraints is called timing closure. This is accomplished through layout optimizations and netlist modifications. These can be: [1]

- *Transistor Resizing:* Altering the width to length ratio of transistors to increase the drive strength of gates or decrease their delay and meet setup constraints
- *Buffer insertion:* Inserts buffers into nets to increase their delay and meet hold constraints
- *Critical path restructure:* Restructures the circuit along its critical paths (e.g decrease the depth of a tree to reduce delay).

To check if setup constraints are met, requires the estimation of the time it takes for a signal to propagate from one storage element to the next. This depends upon gate delays and wire delays. In older technologies where gate delays where the dominating factor, the number of gates on a timing path provided a good estimate. However in today's technologies that is totally inaccurate due to the high importance of wire delays. This adds up to the complexity of the delay calculation. This delay estimation is based on static timing analysis (STA).

Two very important metrics in timing closure are the Required Arrival Time (RAT) and the Actual Arrival Time (AAT). The former is the time by which the latest transition at a given point must occur in order for the circuit to operate correctly within a given clock cycle. The latter is the actual time the signal arrives. Based on these two, a new key metric is derived called slack = RAT - AAT. A positive slack means that a signal arrives before its required and so timing is met. On the other hand a negative slack means that a signal arrives late, after its required time and so timing is violated.Figure 2.16 illustrates the slack definition for a setup constraint which is met.

In hold checks, a positive slack means the exact inverse which means that the signal arrives later than required. From now on, we are exclusively going to discuss setup constraints.

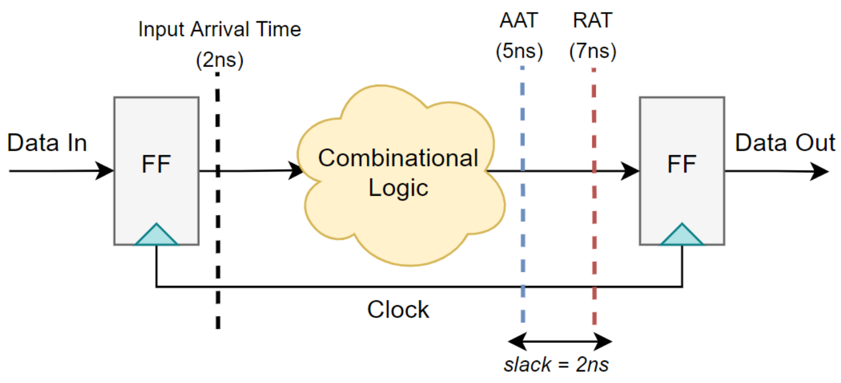


Figure 2.16: An example with the slack definition.

2.4.5 Static Timing Analysis

Estimation of these values is based on an efficient, linear-time verification process called Static Timing Analysis (STA). STA is used to propagate actual arrival times (AATs) and required arrival times (RATs) through every gate of a path to calculate their slack. It identifies timing violations, and diagnoses them by tracing out critical paths that are responsible for these timing failures. Because it's computationally impossible to identify which paths are sensitized (cause a transition) or not, STA only picks the longest paths (i.e paths with longest delay) so it is pessimistic in nature. That is also the reason which is called static, because it does not depend on the current input. The extent to which a design satisfies timing constraints is given by two very important timing metrics. The first is the Worst Negative Slack (WNS) metric which as the name implies is the slack with the most negative value in the design. The second metric is called Total Negative Slack (TNS) and is the summation of every slack with a negative value.

As aforementioned in the previous chapter, any combinational circuit can be represented as an undirected hyper-graph $G = (\mathcal{V}, \mathcal{N})$, where set $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ are the vertices i.e. the circuit components of the design and set $\mathcal{N} = \{n_1, n_2, \ldots, n_n\}$ are hyper-edges i.e. nets that connect them. The primary inputs to the circuit are treated as the outputs of of dummy components in $\mathcal{V}$. Let $n(v)$ denote the output net of component v.

A mapping is defined as:

$$\pi(v, n) = \begin{cases} -1 & \text{if } n \text{ is an input to } v \\ 1 & \text{if } n \text{ is an output to } v \\ 0 & \text{otherwise } (n \text{ not connected to } V) \end{cases}$$

Fan in of component $v \in \mathcal{V}$ is:

$$\pi^-(v) = \left\{ v' \middle| \pi(v, n(v')) = -1 \right\}$$

Fan out of component $v \in \mathcal{V}$ is:

$$\pi^+(v) = \left\{ v' \middle| \pi(v', n(v)) = -1 \right\}$$

Actual Arrival Time (AAT) t each input of component v is calculated as:

$$t_{AAT}(v) = \max_{z \in \pi^-(v)} t_{AAT}(z) + d(v) \tag{2.10}$$

Required Arrival Time (RAT) at each output of component v is calculated as:

$$t_{RAT}(v) = \min_{z \in \pi^+(v)} t_{RAT}(z) - d(z) \tag{2.11}$$

Finally slack is defined as:

$$s(v) = t_{RAT}(v) - t_{AAT}(v)$$

In STA the hyper-graph $G = (\mathcal{V}, \mathcal{N})$ is transformed into a directed acyclic graph (DAG) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where every net $n \in \mathcal{N}$ is represented by edges
$e \in \mathcal{E} \mid n_i = \{e_1, e_2, \ldots, e_n\}$. Every edge represents an electrical connection between the output driver gatepin of a component, and the input pins it connects to. The latter also defines the direction of the graph.

Each component v, is associated with a delay *d(v)*, which represents the time required for a change in signal at one of its inputs to result in a change at the inputs to components in $\pi^+(v)$.

This delay consists of two parts. The first is the delay due to the parasitics of the component v it self, called base delay $d_0(v)$, the second is the delay due to the interconnection between v and $\pi^+(v)$, symbolized as $\delta(v)$, i.e. $d(v) = d_0(v) + \delta(v)$.

Figure 2.17 illustrates a circuit with three inputs $\mathbf{a}, \mathbf{b}, \mathbf{c}$, four combinational logic gates x, y, z, w and one output **f**. Numbers inside the brackets represent the interconnection delays $\delta(v)$ and numbers inside the parentheses the base delays

$d_0(v)$. The inputs are annotated with times in the form: $input\_name < value >$ which represent the signal transition times relative to the start of the clock cycle.

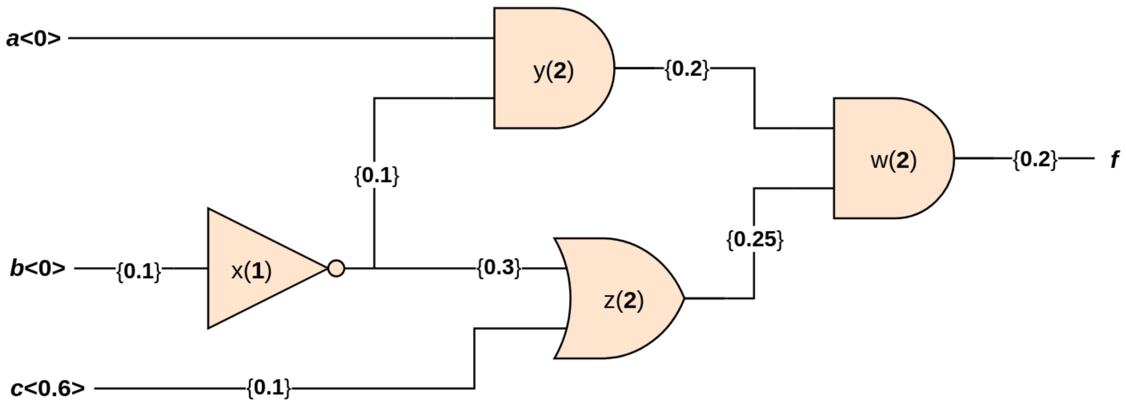


Figure 2.17: Circuit STA example

Figure 2.18 corresponds to the directed acyclic graph, which has one node for each logic gate, as well as one note for each input and output. As mentioned before, inputs can be thought as outputs of dummy cells and nodes $\mathbf{a}, \mathbf{b}, \mathbf{c}$ represent just that. Also for convenience, another dummy cell is introduced, which represents a source node.

At first, a forward traversal of the graph is made to propagate the Actual Arrival Time (AAT) values to each node, based on equation 2.10. At each node v we check the fan-in and pick the AAT with the maximum value. Then we add this value to the delay *d(v)*, i.e. the interconnection delay plus the base delay of the node.

Next, we do a backward traversal of the graph to propagate the Required Arrival Time (RAT) values to each node, based on equation 2.11. At each node v we check the fan-out and pick the RAT with the minimum value. Then we subtract the delay *d(v)* from it.

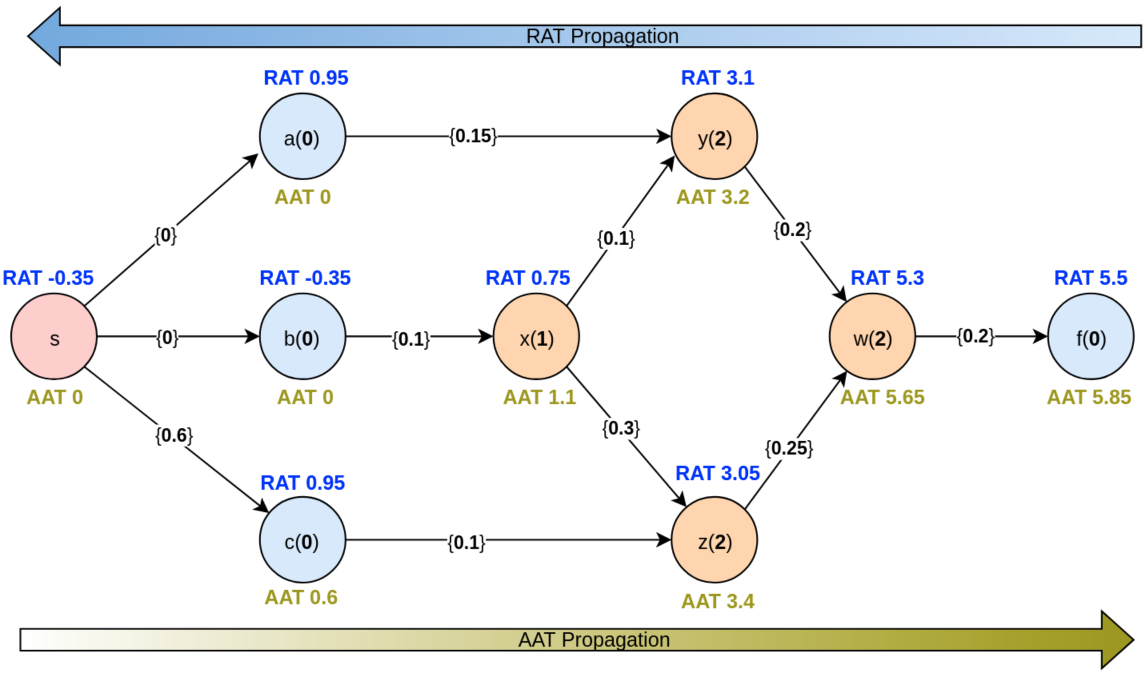


Figure 2.18: Circuit STA example

After the AAT and RAT times are calculated, we proceed to the calculation of the slack at each node which is simply the substraction between the two values. With green we represent slack that meet the timing requirements and with red, the opposite. Red edges illustrate the critical path, i.e. the path which produces the negative slack value at the output **f**.

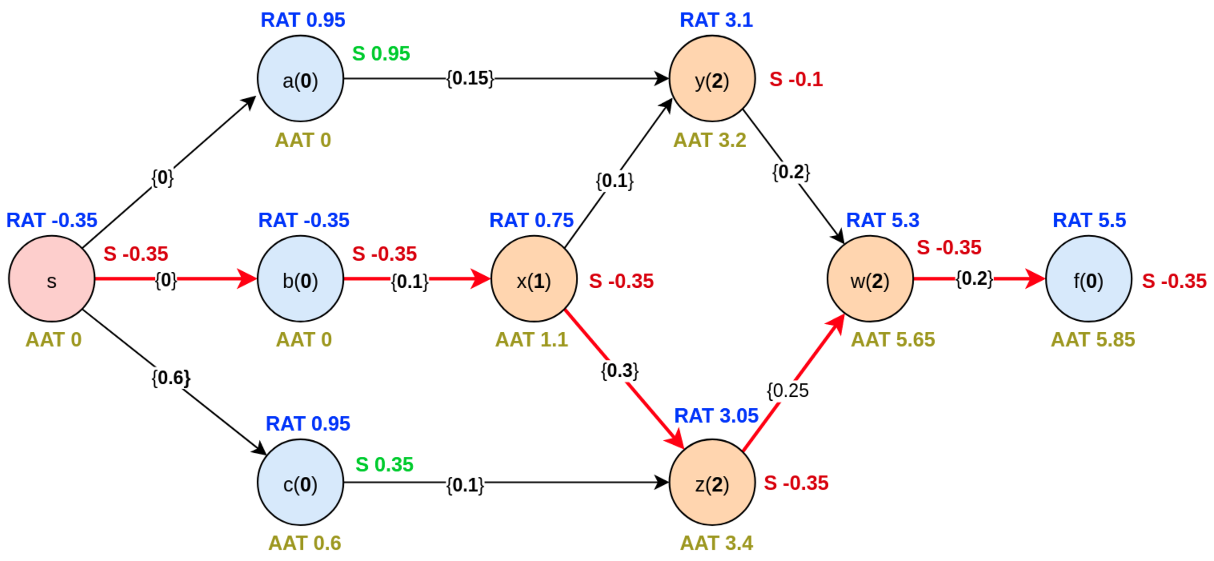


Figure 2.19: Circuit STA example

Chapter 3

Existing Work

3.1 Introduction

In the last decades the area of Timing-Driven Placement has been studied extensively and a variety of methods and techniques exist in the literature. These can be distinguished into two main categories, the net-based and path-based. The former approach handles nets by giving higher priority to critical ones, either by the form of net-weights or net-constrains and formulates the problem as the minimization of the total weighted wirelength to give direction to the placer for shortening critical nets. The latter handles all or a subset of paths directly and attempts to meet their timing requirements simultaneously. The majority of the path based methods, formulate this problem using mathematical optimization like linear programming. Both of these approaches have their pros and cons. Net-approaches have low computational complexity and run time, high flexibility, are easier to implement and they have good scalability to large designs. However due to the fact that they deal with the timing problem indirectly, there is an ambiguity regarding the selection of the weights. Their impact is often unpredictable and varies from design to design. That leads to the need of extensive parameter tuning to adapt on specific design styles. Good weight generation is difficult and a bad one can lead to overconstrained designs with a lot of increase to their total wirelength. Path based approaches cope with the timing problem explicitly so they are more accurate but they suffer from the very high complexity of modeling timing in an exponential growing number of paths and thus scale really bad as designs get larger. Another possibility is the use of a hybrid approach which combines both of these methods and shows promising results. [6]

3.2 Net Based Approaches

Most placement algorithms converge to a proper placement solution by minimizing the total wirelength for all the different nets of the design. Net based approaches add the modification of assigning weights or constraints to nets. We first start by describing the net weighting techniques.

3.2.1 Net Weighting Techniques

Net weighting techniques, assigns weights to nets, considering their criticality. The more critical a net is, the higher its weight. Criticality for a net can be determined from a number of factors e.g. the slack value. Then the placement objective changes to the minimization of the weighted total wirelength for the purpose of directing the placement engine to keep critical connections shorter. The advantage of net weighting is that it can be easily integrated into almost all placement algorithms. For example quadratic placement, optimizes the weighted quadratic wirelength and partitioning based placement, optimizes the weighted cutsize.

There are two main methods of net weighting techniques, the empirical net weighting and the sensitivity net weighting. The former computes weights based on certain criticality factors, such as slack, cycle time and fanout. The latter computes weights based on a type of analysis which tries to estimate the impact of net weights on some key design factors such as WNS and TNS. In other words it estimates how sensitive they are to the weights change. Sensitivity based approach is considered more accurate because it includes a form of look-ahead mechanism that leads to better weight generation and tries to counteract the unpredictability of the net weighting process. However that comes to the expense of being more complex and computationally demanding.

All of these methods can be applied statically or dynamically. Static net-weighting calculates weights once, based on timing information that is available pre-placement and does not update the weights during the placement iterations. Dynamic net-weighting obtain new timing information after a number of placement iterations and weights are adjusted accordingly. It is of course more accurate than static net-weighting, but it needs to be handled more carefully due to some issues like component oscillation. It also adds up significant run time due to the algorithms run to derive the timing information. The most common algorithm to run is Static Timing Analysis (STA) although other methods have also been developed to mitigate the run time penalty.

Empirical Net Weighting

Empirical net weighting generates weights based on the criticality of a net. Criticality is more often measured by slack but other metrics can also be used. After the metric is defined, then a mathematical relationship between it and the weight must be established. This takes the form of a function with the metric as its input and the generated weight as its output. The function maps every value inside the metric domain into a weight. If we chose slack as the metric, the most straightforward approach is to define a linear function in the form of:

$$W(slack) = a * slack + b$$

where a, b constants which are parametrized based on design information. Constant a must be negative so that the slope of W gives higher weights as slack becomes more negative. Fig. 3.1 shows the graph of that function.
Another option is to define a segmented function that gives even more priority to

critical nets in the following way:

$$W(slack) = \begin{cases} w1(slack) & slack \leq 0 \\ w2(slack) & slack \geq 0 \end{cases}$$

The $w1$, $w2$ are linear functions with different gradients separated by the zero point. Zero is what distinguishes between critical (upper segment) and non-critical (lower segment) nets, where $w1 > w2$ at all times. Fig. 3.2 represents the W function.

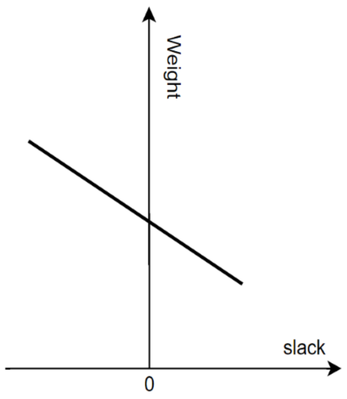


Figure 3.1: Linear mapping

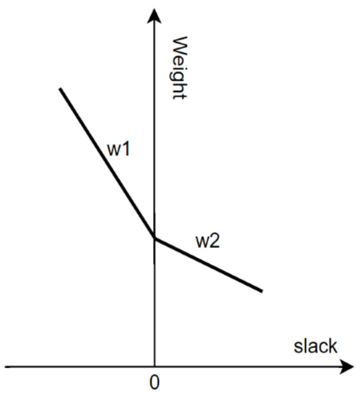


Figure 3.2: Segmented mapping

Two other metrics that can be used are the net path depth and net driver strength. Net path depth is defined as the longest path that can be discovered starting from the beginning of that net. A design with more depth and thus lots of stages is more likely to have more delay and more wirelength. Also a net with a weak driver will have a longer delay as it would take more time to charge the various parasitic capacitances. By combining these two, weight is proportional to the longest path depth and inversely proportional to driver strength of a net (proportional to driver resistance):

$$W \approx D_l \times R_d$$

Another approach found in the literature derives a metric called path sharing. Inherently, a net which is present on many critical paths should be assigned a higher weight because reducing the length of such a net can reduce the delay on many critical paths. In [7] an algorithm is developed called Path Counting Based Weighting Algorithm (PATH) which calculates the total number of critical paths that pass a net (*path_sharing*). Then a weight is extracted as:

$$W \approx slack \times path\_sharing$$

Following is a method [8] that extracts weights without using placement information i.e pre-placement. Before placement, the locations of modules are unknown. The only available information includes a cell library and the design's netlist. Then weights are extracted in the following way:

$$W \approx \frac{fanout}{netdelaybound}$$

Fanout can be a good estimation of wirelength and delay. Delay bound is a metric that estimates the allowable wire delay, any delay above this bound will result in a negative slack. It is computed by and algorithm called IMP which utilises the timing requirements of the circuit and the switching delays of cells [9]. Pre-placement weight extraction has the advantage of having negligible increase of runtime because of the lack of dynamic methods to obtain timing information during the placement iteration steps.

Sensitivity Net Weighting

Although net-weighting might improve timing on critical paths, it can have negative impact on the total wirelength of the design. An aggressive high weight assignment reduces the connections between critical components by moving them closer together but at the same time increases other connections which are common to these components. Net-weighting timing driven placers have a trade-off between an improvement in timing, and an increase in total wirelength with small exceptions in some specific designs.

Sensitivity based weighting seeks to find those connections that if they get shortened, will have the maximum impact on timing. On the other hand connections that don't improve timing significantly, get smaller weights to minimize the penalty of the increased wirelength. Specifically it tries to solve the following problem: If we increase the weight of a net *i* by a certain amount, how much improvement net *i* will get fot its worst negative slack (WNS) and its total negative slack (TNS)? In other words, how sensitive WNS and TNS are to the net weight change? After answering the above question the weight of a net can be determined by the following formula: [10]

$$W(i) = \begin{cases} W_{init}(i) & Slk(i) \geq Slk_t \\ W_{init}(i) + \Delta W^*(i) & Slk(i) \leq Slk_t \end{cases} \tag{3.1}$$

where

$W_{init}(i)$: The initial weight of net i

$$\Delta W^*(i) = \alpha[Slk_t - Slk(i)]S_W^{Slk}(i) + \beta S_W^{TNS}(i)$$

and

- $S_W^{Slk}(i)$: Sensitivity of slack to weight, thus how sensitive the slack is to the weight change.
- $S_W^{TNS}(i)$: Sensitivity of TNS to weight, thus how sensitive the TNS is to the weight change.
- α, β are constants that define the weight upper and lower bound values and control the balance between WNS and TNS reduction.

For non-critical nets the weight remains the same. For critical nets the weight increase depends on how much that weight is able to reduce the WNS and TNS metrics. If the sensitivity is low then there is no need to increase the weight too much, because it will have negative impact by increasing the overall wirelength. On the other hand if the sensitivity is high, then larger weight values can be utilized to have the maximum impact possible on the negative slack reduction.

From a bottom up perspective to estimate $S_W^{Slk}(i)$ and $S_W^{TNS}(i)$ first we need to estimate the impact of weight change to wirelength, i.e $S_W^L(i)$, then the impact of wirelength on delay, i.e $S_L^T(i)$ and finally obtain:

$$S_W^{Slk}(i) = -S_L^T(i) S_W^L(i) \tag{3.2}$$

and

$$S_W^{TNS}(i) = -K(i) S_W^{Slk}(i) \tag{3.3}$$

where $K(i) = \frac{\Delta TNS}{\Delta T(i)}$, which means how sensitive total negative slack (TNS) is to a change, if the delay of a net i changes. This is proven to be equal to the negative of the number of critical timing endpoints whose slacks are influenced by net *i*. It can be computed by a specific algorithm which is omitted as its explanation is beyond the scope of this thesis.

The estimation of the weight to wirelength sensitivity for a net i is defined as follows:

$$S_W^L(i) = \frac{\Delta L(i)}{\Delta W(i)} = -L(i) \cdot \frac{W_{src}(i) + W_{sink}(i) - 2W(i)}{W_{src}(i) W_{sink}(i)} \tag{3.4}$$

where

- *L(i)* is the current wirelength of net i
- *W(i)* is the current weight of net i
- $W_{src}(i)$ is the summation of all the net weights that intersect with the driver (source) of the net
- $W_{sink}(i)$ is the summation of all the net weights that intersect with the receiver (sink) of the net.

All these are defined for the sample circuit shown in Fig. 3.3 which shows a net defined between two components *i*, *j*, nets defined between the driver i and other components and nets defined between the receiver *j* and other components. For the sake of simplicity net (*i*, *j*) is denoted as net *i*.

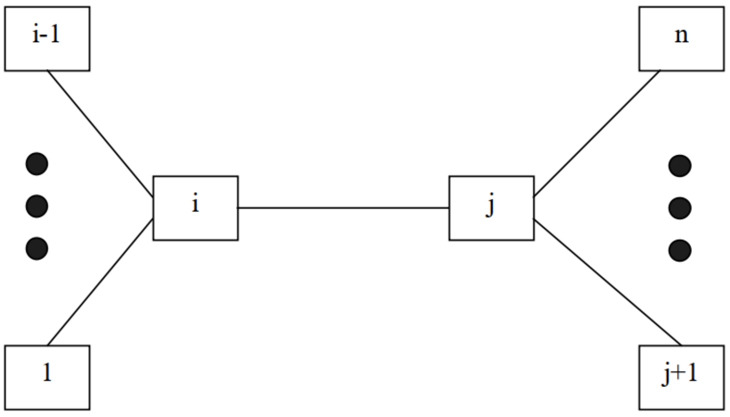


Figure 3.3: Sample circuit for wirelength to net weight sensitivity.

Equation 3.4 implies that if the weight of a net changes by a certain amount, then the impact it has on the wirelength of the net, is relative to the current wirelength of the net *L(i)*. If a net has big wirelength it is more sensitive to the weight change. On the other hand if a net has small wirelength then weight won't affect it much. Furthermore if the current weight of a net *W(i)* is big, then increasing it more won't have much impact. Inversely a net with small weight, is expected to have big impact on the wirelength if it is increased.

Next the definition of the wirelength to delay sensitivity is given as:

$$S_L^T(i) = \frac{\Delta T(i)}{\Delta L(i)} = rcL(i) + cR_d + rC_l \tag{3.5}$$

where

- r and c are the unit length wire resistance and capacitance, respectively
- R_d is the output resistance of the net driver (*i*)
- C_l is the output capacitance of the net receiver (*j*)

Equation 3.5 implies that for a given technology (fixed *r* and *c*), the longer the net, the weaker the driver (larger R_d) and the larger the load seen by the net is, then the more the delay increases for a certain amount of wirelength change.

With equations 3.4 and 3.5 defined, then $S_W^{Slk}(i)$ and $S_W^{TNS}(i)$ can be easily calculated from the equations 3.2, 3.3.

Incremental Net Weighting

Dynamic net-weighting although more accurate than the static one, it might cause convergence issues if it is not handled carefully. The following scenario can occur: Critical nets are assigned high weights. In the next placement iteration these nets reduce their length but simultaneously non-critical nets increase theirs. Timing analysis runs again and the non-critical nets can now become critical due to the length increase. This situation where a net alternates between being critical and non-critical can remain and cause oscillations. To mitigate this issue and make timing-driven placement more stable an approach called incremental net weighting can be used.

Incremental net weighting keeps a history database for every net of the design containing information which tells if a net was critical or not in previous placement iterations. If a net was critical before there is a high change that it will become critical again if its weight is reduced. So to avoid this scenario, this method reduces the weights of non-critical nets with a critical history more smoothly.

Below a method [11] is shown of calculating a weight of a net *i*, on placement iteration *k*, by keeping a two iteration history of a metric called criticality, c_i^k:

$$w_i^k = \begin{cases} w_i^{k-1} + W & \text{if } c_i^k = 1 \\ 1 & \text{if } c_i^k = 0 \wedge c_i^{k-1} = 0 \wedge c_i^{k-2} = 0 \\ \lceil w_i^{k-1}/2 \rceil & \text{if } c_i^k = 0 \wedge c_i^{k-1} = 0 \wedge c_i^{k-2} = 1 \\ w_i^{k-1} & \text{if } c_i^k = 0 \wedge c_i^{k-1} = 1 \end{cases} \tag{3.6}$$

Criticality is either 1 or 0 to define if the net is critical or non-critical respectively. In this method, if a net is critical in the current iteration, then its weight is increased by a factor W. If a net is non-critical in the current iteration and was also non-critical in the two previous iterations then weight is assigned the neutral value of 1. If a net is non-critical in the current iteration, but it was critical two iterations before, then weight is divided by two. If a net is non-critical in this iteration but it was critical just on the previous iteration, then the weight stays the same.

3.2.2 Net Constraint Techniques

Zero Slack Algorithm

As already mentioned, net weights suffer from the ambiguity regarding their selection due to the fact that they do not have an immediate connection with the timing space. Net constraint methods on the other hand, are more natural choice, because there is a direct and computable correspondence between a net length and the delay of that net.

Another problem which appears frequently in net-weighting methods is that, in attempting to minimize the length of all nets along a critical path, the problem is often overconstrained to the point where nets which were not critical become excessively long. This can be avoided by using net constraint methods.

Net constraint techniques relay on timing information to extract the constraints. The most commonly used timing information, is the delay bound which corresponds to the maximum allowable delay a net can have until it violates its setup requirements. A popular method for calculating delay bounds on each net, is the Zero Slack Algorithm (ZSA) [12] and this section starts with its description.

The Zero Slack Algorithm , generates upper bound delays on each net of a combinational circuit having specified timing requirements at its input and output terminals.

Each component x, is associated with a delay $d(x)$, which represents the time required for a change in signal at one of its inputs to result in a change at the inputs to components in $\pi^+(x)$.

This delay consists of two parts. The first is the delay due to the parasitics of the component x it self, called base delay $d_0(x)$, the second is the delay due to the interconnection between x and $\pi^+(x)$, symbolized as $\delta(x)$. The difference,

$$\delta(x) = d(x) - d_0(x)$$

is the contribution to delay due to the net $e(x)$.

The Zero Slack Algorithm seeks to decrease positive slacks of all nodes to zero by determining a proper set $\Delta(\mathcal{X}) = \{\delta(x_1), \delta(x_2), \ldots, \delta(x_n)\}$. The former is called the timing budget set and represents the maximum allowable net delays before the slack of the circuit turns to negative. These timing budgets can then be transformed into wirelength budgets (constraints) and drive the placement algorithms appropriately.

Algorithm 1 presents the pseudocode implementation of ZSA. The algorithm can be divided into three major steps repeating until the slack of the output components is equal to zero.

1. The initial slacks of all the circuit's components represented by graph G, are determined. This can be accomplished with Static Timing Analysis. Also in that same step, the minimum positive slack s_{min} of the graph is calculated.
2. A *path* is determined that starts from the component with the minimum slack x_{min}. This *path* includes all the components that both their RAT and their AAT is calculated based on x_{min} by exploring the predecessor and successor directions. This means that all the components of *path*, will have the same slack equal to s_{min}.
3. Delay bound is increased uniformly to every component of the *path* and it is proven that each of their slack gets equal to zero. Uniform distribution is important to avoid situations where the delay bound is lumped with only one component leaving the other components of the *path* unconstrained. The goal is to constrain as much nets as possible to aid the efficiency of the global placement algorithms.

After the completion of the above steps, slacks are recalculated with Static Timing Analysis and the procedure repeats.

In figure 3.5 an example demonstrates how the algorithm works. Every component is represented as shown in figure 3.5. At the circuit shown, output $G2$ is critical in a sense that no delay bound can be specified without producing a negative slack. Output $G1$ has a positive slack = 4, and delay bounds can be specified to drop it to zero. First the component with the minimum positive slack of the design is specified. That is component 3. Then *path* is specified which includes all the components with that same slack, by traversing in both the predecessor and successor directions. After that, component 5 is found and *path* = $\{3, 5\}$. Then slack is distributed between components 3, 5 as, $s = s_{min}/|path| = 2/2 = 1$ and delay bounds are $\delta(3) = \delta(4) = 0 + s = 1$. Then timing information is updated again, by running STA and the algorithm keeps iterating until slack of output $G1$ is equal to zero. The output result of the algorithm is $\Delta(\mathcal{X}) = \{2, 0, 1, 0, 1, 0, 2, 0\}$.

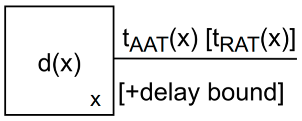


Figure 3.4: ZSA metrics representation

The key takeaways of the above procedure are:

- Delays are distributed along components that have the minimum slack value (members of *path*).
- The sum of incremental delays introduced on these components is exactly equal to that minimum slack.
- The slack on each of these components is exactly equal to zero.
- The circuit remains safe at the end of each iteration i.e. no negative slack is produces at the output gatepins.

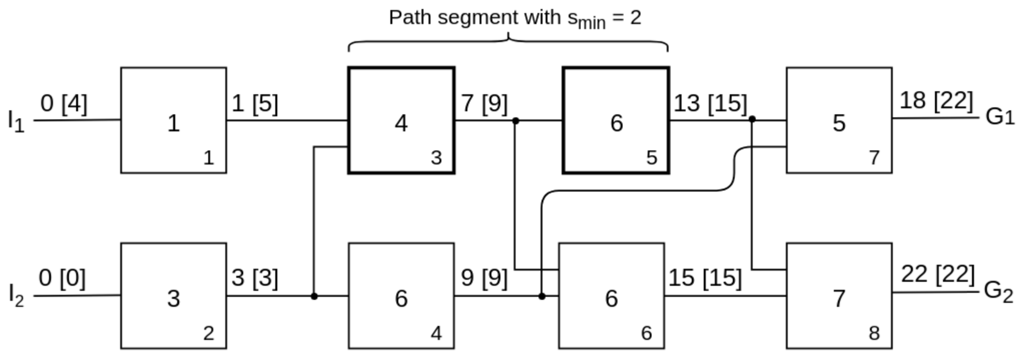


Figure 3.5: ZSA first iteration

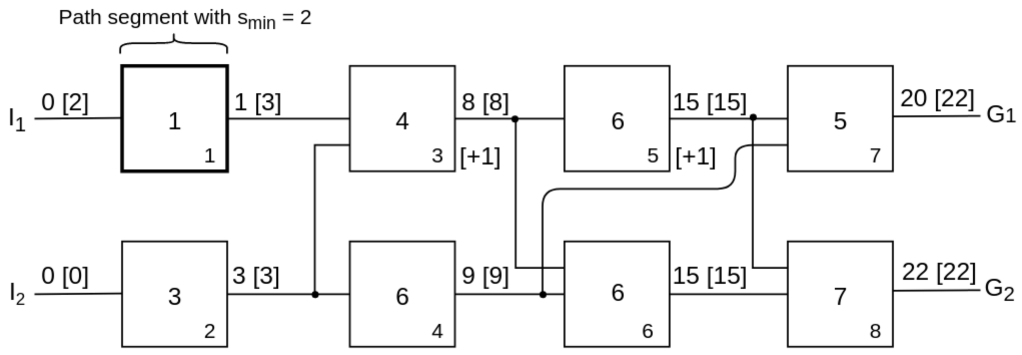


Figure 3.6: ZSA second iteration

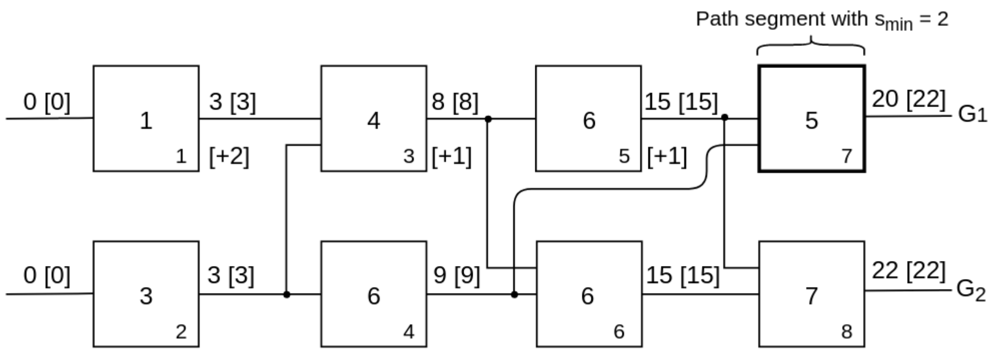


Figure 3.7: ZSA third iteration

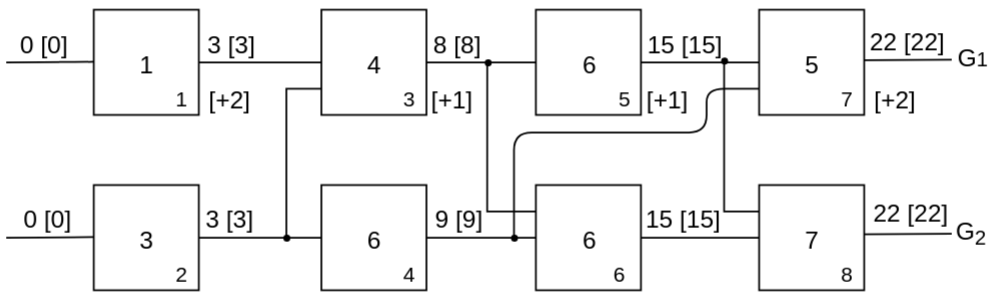


Figure 3.8: ZSA finalized result

Algorithm 1: Zero Slack Algorithm

```
   input  : $G = (\mathcal{X}, \mathcal{N})$
   output: $\Delta(\mathcal{X}) = \{\delta(x_1), \delta(x_2), \ldots, \delta(x_n)\}$
 1 repeat
       // Compute slacks using STA
 2     $(AAT, RAT, slack) = STA(G)$;
       // Compute minimum positive slack
 3     $s_{min} = \infty$ ;
 4     foreach $x_i \in \mathcal{X}$ do
 5         if $s[x_i] < s_{min}$ and $s[x_i] > 0$ then
 6             $s_{min} = s[x_i]$;
 7             $x_{min} = x_i$
 8         end
 9         if $s_{min} \neq \infty$ then
               // Find forward path segment
10             $u = min$;
11             $path[u] = x_u$;
12             while $x \in \pi^+(x_u) \mid t_{RAT}(x) = t_{RAT}(x_u) + d(x)$ and
                $t_{AAT}(x) = t_{AAT}(x_u) + d(x)$ do
13                 $u{+}{+}$ // go forward;
14                 $path[u] = x$;
15             end
               // Find backward path segment
16             $u = min$;
17             $path[u] = x_u$;
18             while $x \in \pi^-(x_u) \mid t_{RAT}(x) = t_{RAT}(x_u) - d(u)$ and
                $t_{AAT}(x) = t_{AAT}(x_u) - d(u)$ do
19                 $u{-}{-}$ // go backward;
20                 $path[u] = x$;
21             end
               // Distribute slacks
22             $s = s_{min}/|path|$;
23             for $i = u$ to $|path|$ do
24                 $\delta(i) = \delta(i) + s$ // increase net delay;
25             end
26         end
27     end
28 until $slack_{min} \neq \infty$;
```

Physical Net Constraints on Timing Driven Force Directed Placement

Here we present a method [13] which sets half perimeter wirelength constraints on critical nets. It builds on the work of the Kraftwerk [2] force directed algorithm. It adds a new net model which changes the contribution of constrained nets in the quadratic cost function at each placement iteration.

Figure 3.9 depicts how this process works. When a net surpasses the constraint value c a weight is generated corresponding to that net. This weight can be distributed to all the edges of a net or more sophisticated methods can be used [13]. The weight increases linearly with the increase of the HPWL until it reaches an upper bound value. This value is used to avoid over-constraining the design. Very critical nets won't necessarily improve with further increasing the weights and new critical nets might appear.

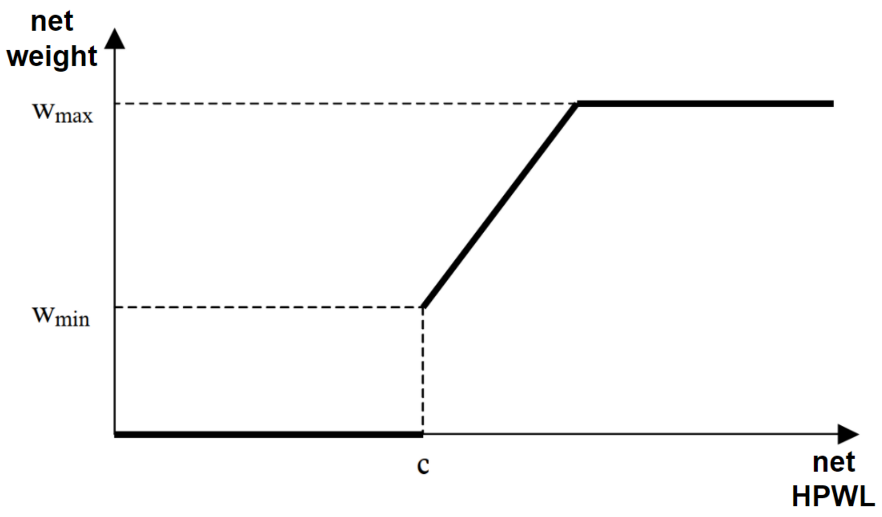


Figure 3.9: Additional net weight w derived based on the amount of the HPWL of the net exceeding the net constraint c

When a weight is assigned to a net, it is maintained even if the net is no longer violating the constraint. This means that weight can only increase through subsequent placement iterations. This is done to avoid the scenario where this net, due to the relaxed net-force, becomes a violator again.

Net constraints are only specified for critical nets, and these nets are identified based on heuristic methods and considering criticality factors such as slack and driver strength as discussed in the previous chapter.

Constraints can be generated using the delay bounds extracted from the ZSA algorithm or different timing metrics, or they can be totally heuristic.

3.3 Path Based Approaches

Path based approaches refer to the algorithms which directly model the timing constraints during placement. Their advantage is the fact that they handle the timing problem directly because they are not based on heuristics such as net weights. Their disadvantage is the very large number of paths which makes the their timing modeling unfeasible. For this reason most of the timing driven work in the literature is net based. Here we give the general problem formulation of the path based approaches.

A technique solves the high number of paths problem by formulating it into a Linear Programming (LP) optimization problem [14]. It embeds a timing graph into the Timing Driven Placement formulation, based on a simplified version of Static Timing Analysis. The flow begins with a given placement and iteratively extracts timing-critical sub-circuits and creates objective functions to optimize. These can be the HPWL minimization of critical nets or the maximization of the TNS (or WNS) value. This approach however is notorious for producing a lot of overlap and needs good handling by a legalizer.

The general form of the objective function is the following:

$$\text{maximize/minimize} \quad f(\mathbf{X}) \tag{3.7a}$$
$$\text{subject to} \quad A\mathbf{X} \leq D \tag{3.7b}$$

$\mathbf{X}$ is a set of variables including cell coordinates and auxiliary variables.
$f(\mathbf{X})$ can be any of the aforementioned cost functions.
$A\mathbf{X} \leq D$ is any physical or electrical constraints such as HPWL constraints, delay constraints, slack constraints, etc.

An example of an objective function derived by [14] is:

$$\text{maximize} \quad TNS \tag{3.8a}$$
$$\text{subject to} \quad HPWL \leq HPWL_{limit} \tag{3.8b}$$

which gives good timing results without suffering from excessive wirelength increase.

Chapter 4

Timing Driven Placement

In this chapter we describe the implementation of our net weight assignment method. We first start by describing the general details of our Global Placer, which is based on a very popular force-directed algorithm called Kraftwerk2 [15]. This part describes, the different forces that act on components in detail and it's essential for understanding our timing driven implementation.

Next we describe our net weight assignment process by presenting the general notion behind it, the different mapping functions used, the way they translate timing information into weights and the way weights are distributed into the circuit.

Finally we present how the above, is linked with the Kraftwerk2 algorithm. The general overview of the Timing Driven Global Placement is shown in figure 4.1. Weights act as an interface between the timing analysis and the core placement algorithms. Also there is a feedback between the placement layout and the timing analysis at every iteration, to update timing information.

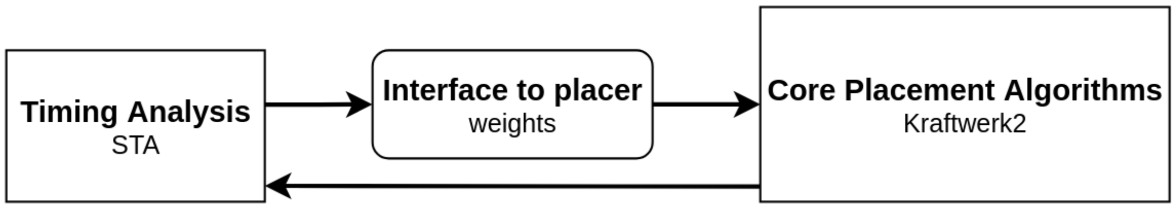


Figure 4.1: Timing Driven GP overview

4.1 Kraftwerk2 Algorithm

Kraftwerk2 [15] is an analytic quadratic global placer which formulates the placement problem with a set of forces that act on cells. The first force, is called net force and corresponds to the spring force from Hooks Law. Its purpose is to minimize the quadratic cost function, by keeping components close together. The magnitude of this force is proportional to the distance between the components. The second force, is called move force and its purpose is to remove overlap between components. The magnitude of this force is proportional to the amount of overlap at each point inside the core area. Both of these forces act simultaneously and iteratively, leading the global placement problem to a solution convergence at the event of a force equilibrium. Additionally kraftwerk2 describes a new net model called "Bound2Bound" which exactly represents the half-perimeter wirelength (HPWL) in the quadratic cost function and achieves better performance than conventional net models.

4.1.1 Problem Formulation and Bound2Bound net model

In the case of quadratic placement, each net $n \in \mathcal{N}$, is represented by a set of two-pin edges $e \in \mathcal{E}_n$, which connect the pins of the net. One edge is defined as $e = (p, q)$ and connects pins p and q. The netlength of the circuit Γ is expressed in the following quadratic cost function:

$$\Gamma(\mathbf{x}, \mathbf{y}) = \frac{1}{2} \sum_{n \in \mathcal{N}} \sum_{e \in \mathcal{E}_n} c_{x,pq}(x_p - x_q)^2 + c_{y,pq}(y_p - y_q)^2 \tag{4.1}$$

Net models simply specify the set $\mathcal{E}_n$, i.e. which edges represent the net $\mathcal{N}$. If the clique net model is used, all possible two-pin connections of the net are utilized and the cost function becomes:

$$\Gamma(\mathbf{x}, \mathbf{y}) = \frac{1}{2} \sum_{p=1} \sum_{q=p+1} c_{x,pq}(x_p - x_q)^2 + c_{y,pq}(y_p - y_q)^2 \tag{4.2}$$

Connection costs c_x, c_y can be utilized to approximate different net models. For example GordianL uses the following formula to adapt the clique model to the star model:

$$c_{x,pq}^{GordianL} = \frac{1}{P}\frac{2}{P}\frac{4}{|x_p - x_q|} \qquad c_{y,pq}^{GordianL} = \frac{1}{P}\frac{2}{P}\frac{4}{|y_p - y_q|}$$

The first two factors adapt the clique model to the star model and the third factor linearizes the quadratic distance between pins p and q.

In the clique net model, there is a high approximation error between the length of the clique net and the HPWL. That is due to the existence of connections between inner pins and their lengths, which contribute to the clique length but are ignored in the HPWL metric.

Due to the above the *Bound2Bound* net model is developed which achieves the following:

- Linearizes the cost function
- Has lower connections at each net than the clique net model, which leads to more sparse matrices and better run times.
- Does not require extra pins like the star net model.
- Exactly represents the HPWL metric.
- Achieves better minimization of the given objective than the clique and hybrid clique/star net models.

The *Bound2Bound* net model works by removing all inner two-pin connections and utilizing only connections to the boundary pins. Figure 4.2 shows a circuit with a 5-pin net defined between an output yellow pin and four blue input pins.

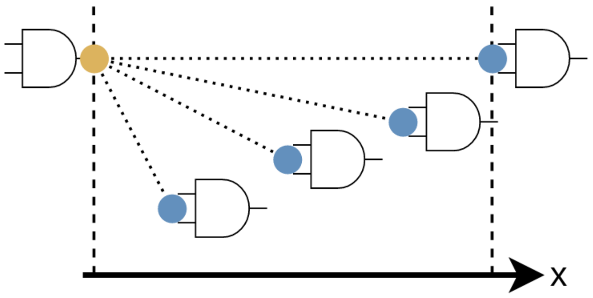


Figure 4.2: Circuit defining a 5-pin net

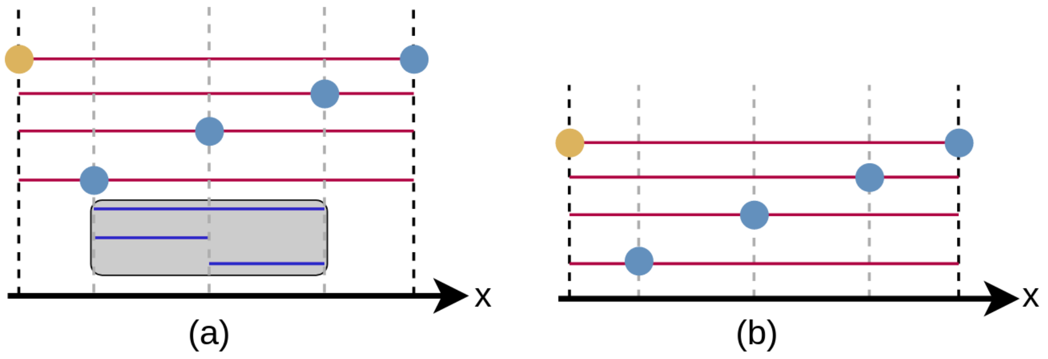


Figure 4.3: Clique net model (a) and Bound2Bound net model (b)

Figure 4.3 (a) illustrates the clique edges for the x direction. Edges between input pins (shown inside the gray area) are removed. By utilizing only connections to the boundary pins the property of the HPWL netlength is emulated.

The connection weight of the *Bound2Bound* net model is given as follows:

$$c_x^{B2B} = \begin{cases} 0 & \text{p,q inner pins} \\ \frac{2}{P-1}\frac{1}{|x_p - x_q|} & \text{else} \end{cases} \tag{4.3}$$

It is mathematically proven that if 4.3 is used into the cost function 4.1, it exactly represents the HPWL for each net.

Also the *Bound2Bound* net model, linearizes the cost function. Each connection inside the cost function, is evaluated as:

$$c_{x,pq}(x_p - x_q)^2$$

If we apply the *Bound2Bound* net model we achieve,

$$c_x^{B2B} \cdot c_{x,pq}(x_p - x_q)^2 = \frac{2}{P-1} c_{x,pq}(x_p - x_q) \tag{4.4}$$

This is very helpful because the minimization of the cost function can be handled directly by solving the system of linear equations and it does not require the calculation of the gradient as shown in the next section.

4.1.2 Cost function minimization and net force

As seen in section 2.3 the quadratic cost function can be separated in the x and y direction. If we focus on the x direction the cost function can be written in a matrix to vector notation as:

$$\Gamma_x(\mathbf{x}) = \frac{1}{2}\mathbf{x^T L_x x} + \mathbf{b_x^T x} + const \tag{4.5}$$

The Laplacian matrix $\mathbf{L_x}$ is a symmetric positive definite matrix which is equivalent to the Γ_x cost function being convex. The minimum is obtained by setting the derivative to zero, i.e.

$$\nabla\Gamma_x(\mathbf{x}) = \mathbf{L_x x} + \mathbf{b_x} = \vec{0} \tag{4.6}$$

Solving the above system of linear equations with respect to $\mathbf{x}$ gives the x-positions of the modules with minimal netlength.

Equation (4.6) is similar to the equation from Hooke's Law:
$\vec{F} = k \cdot x$, Kraftwerk2 introduced the idea of resembling each two-pin connection as an elastic spring that pulls components together. This is called the net-force and is proportional to the length of the net. With that in mind, equation 4.6 can be written as:

$$\mathbf{F_x^{net}}(\mathbf{x}) = \nabla\Gamma_x(\mathbf{x}) = \mathbf{L_x x} + \mathbf{b_x} \tag{4.7}$$

which is a vector that stores the summation of net-forces that act on each component.

Because the integral of the force is energy, this means that the cost function Γ_x represents the energy of the spring system. The problem now can be expressed as the determination of the x-positions that produce the minimum energy state of the system.

4.1.3 Overlap confrontation and move force

If the net force solely acts on the modules, then these are pulled very close to each other and produce a lot of overlap. To treat this issue, force-directed quadratic placers utilize an additional force called move force, with the purpose of pushing modules away from each other into overlap free areas.

To determine the move force, a demand and supply system D is structured, which represents the module overlap and the chip area respectively.

$$D(x, y) = D_{dem}(x, y) - D_{sup}(x, y) \tag{4.8}$$

Equation (4.8) derives a value which quantifies the overlap amount at point (x, y). The greater this value is, the greater the magnitude of the move force exerted on components that lie at point (x, y).

The demand of the system $D_{dem}(x, y)$, is formulated by utilizing the formula of the following function:

$$R_{dem,i}(x, y) = \begin{cases} 1 & \text{if } x, y \in A_{mod,i} \\ 0 & \text{else} \end{cases}$$

This is a mapping which defines if a point (x, y) lies inside an area occupied by module i or not. This area is specified as, $A_{mod,i} = w_i \cdot h_i$, where w_i, h_i is the width and height of the module i respectively.

Then the demand that module i contributes at point (x, y) is calculated as:

$$D_{dem}^{mod,i}(x, y) = d_{mod,i} \cdot R_{dem,i}(x, y) \tag{4.9}$$

where $d_{mod,i}$ is the module density. Module density is normally one but it can be increased or decreased to scale modules up or down respectively. This is helpful for some tuning techniques that can be applied to the Kraftwerk2 algorithm such as removing unwanted free space, around large modules.

Finally the total demand of the system $D_{dem}(x, y)$, is simply the summation of the demands corresponding to each module i:

$$D_{dem}(x, y) = \sum_{i}^{n} D_{dem}^{mod,i}(x, y) \tag{4.10}$$

If $d_{mod,i} = 1$ the value of $D_{dem,i}$ at point (x, y) reflects the number of modules covering this point i.e. how many modules overlap at this point.

The supply of the system $D_{sup}(x, y)$ is formulated in a similar manner. First a mapping is done which defines if a point lies inside the area of the chip A_{chip}:

$$R_{sup}(x, y) = \begin{cases} 1 & \text{if } x, y \in A_{chip} \\ 0 & \text{else} \end{cases}$$

Next the supply of the system D_{sup} is given as:

$$D_{sup}(x, y) = d_{sup} \cdot R_{sup}(x, y) \tag{4.11}$$

where d_{sup} is the supply density. If $d_{sup} = 1$ then equation (4.11) is always 1 for every point that lies inside the chip. This means that the value of the demand and supply system D, eq: (4.8) at a point (x, y) is equal to -1 if free space exists, 0 if that point is occupied by exactly one module and positive if that point is occupied by more than one modules.

Unlike the module density d_{mod}, the supply density is not one by default. Instead its value is calculated based on a formula which is derived by the following analysis.

As a prerequisite, the supply and demand system has to be balanced, i.e. the integral over the demand has to equal the integral over the supply. This is necessary to adapt the demand completely to the supply:

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} D_{sup}(x, y)\, dx\, dy = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} D_{dem}(x, y)\, dx\, dy \tag{4.12a}$$

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} d_{sup} \cdot R_{sup}(x, y)\, dx\, dy = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \sum_{i}^{n} d_{mod,i} R_{dem,i}(x, y)\, dx\, dy \tag{4.12b}$$

$$d_{sup} \cdot A_{chip} = \sum_{i}^{n} d_{mod,i} A_{mod,i} \tag{4.12c}$$

$$d_{sup} = \frac{A_{mod,total}}{A_{chip}} \tag{4.12d}$$

Equation (4.12c) denotes that the area of the chip multiplied by a constant must be equal to the summation of the areas of each module of the chip i.e. the modules have to exactly fit in the $d_{sup} \cdot A_{chip}$ space.

The demand and supply system value $D(x, y)$ can be interpreted as a charge distribution which creates an electrostatic potential $\Phi(x, y)$. This potential is derived by solving the following Poisson's equation:

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)\Phi(x, y) = -D(x, y) \tag{4.13}$$

After determining $\Phi(x, y)$, then the electric field produced by the charge distribution is specified as:

$$\mathbf{E}(x, y) = \nabla\Phi(x, y)$$

From electromagnetic theory we know that electric force is the multiplication of the charge with the electric field. As mentioned before, we can handle the x and y directions separately. By working on the former the move force is specified as:

$$\mathbf{F}_x^{move}(x) = q_{charge,x} \cdot \mathbf{E}(x) \tag{4.14}$$

To be in agreement with the net force F_{net}, which denotes a force produced by a spring connection, we have to represent (4.14) in the form of Hooke's Law. To achieve that, q_{charge} is interpreted as a spring constant $\mathring{w}_i$ and the electric field $\mathbf{E}$, as the distance between a target point $\mathring{x}_i$ and the position x_i of module i.

$$\mathbf{F}_x^{move}(x_i) = \mathring{w}_i \cdot (x_i - \mathring{x}_i) \tag{4.15}$$

If x_i' denotes the module's i position at the beginning of a placement iteration, then the target point $\mathring{x}_i$ can be calculated from (4.15) in the following way:

$$\mathring{x}_i = x_i' - \mathbf{E}(x_i') = x_i' - \frac{\partial}{\partial x}\Phi(x, y)\bigg|_{(x_i', y_i')} \tag{4.16}$$

After the target point is specified, equation (4.15) is used with x_i being the independent variable (also participating into the net-force equation (4.7)) which represents the solution at the end of the iteration.

The spring constant $\mathring{w}_i$ affects the distance the module i moves at each placement iteration. Target point $\mathring{x}_i$ is the upper bound the module i can reach if net-force didn't exist.

Move force is next represented in a matrix-vector notation by collecting the spring constants in the diagonal matrix $\mathring{\mathbf{C}}_\mathbf{x} = diag(\mathring{w}_i)$ and the gradients of the potential in the vector

$$\Phi_x = [(\partial/\partial x)\Phi(x, y)|_{(x_1', y_1')}, (\partial/\partial x)\Phi(x, y)|_{(x_2', y_2')}, \ldots, (\partial/\partial x)\Phi(x, y)|_{(x_n', y_n')}]^T$$

Target points are represented in the vector $\mathring{\mathbf{x}} = \mathbf{x}' - \Phi_x$. Therefore, the move force in matrix to vector notation is

$$F_x^{move} = \mathring{\mathbf{C}}_\mathbf{x}(\mathbf{x} - \mathring{\mathbf{x}})$$

4.1.4 Hold Force and the combined Force-Equilibrium Solution

The kraftwerk2 algorithm works in an iterative manner. Besides the move force, the net force is also acting on the modules and minimizes the netlength. If net-force is not compensated during the beginning of each placement iteration, the modules would collapse back to a position of low netlength and high overlap. This creates oscillations and convergence to a solution is impossible. For this purpose, another force is used called hold force. Hold force is simply the inverse of the net force at the beginning of each placement iteration

$$F_x^{hold} = -(\mathbf{L}_x\mathbf{x}' + \mathbf{b}_x) \tag{4.17}$$

Where $\mathbf{x}'$ are the module positions at the beginning of each placement iteration. Hold force ensures that components spread out evenly at each iteration and they do not return back, to high overlap positions. It is also important to note that the hold force *only depends* on the initial module position $\mathbf{x}'$ and not on the positions during the placement iteration $\mathbf{x}$. Thus hold force, is constant for the duration of one placement iteration.

In summary, the following three forces are used by Kraftwerk2, the net force $\mathbf{F}_x^{net}$, the move force $\mathbf{F}_x^{move}$ and the hold force $\mathbf{F}_x^{hold}$. Setting the sum of these three forces to zero gives the core system of linear equations used in the algorithm's iterative placement process:

$$\mathbf{F}_x^{net} + \mathbf{F}_x^{move} + \mathbf{F}_x^{hold} = \vec{0} \tag{4.18a}$$

$$\mathbf{L}_x\mathbf{x} + \mathbf{d}_x + \overset{\circ}{\mathbf{C}}_\mathbf{x}(\mathbf{x} - \overset{\circ}{\mathbf{x}}) - (\mathbf{L}_x\mathbf{x}' + \mathbf{d}_x) = \vec{0} \tag{4.18b}$$

$$\mathbf{L}_x(\mathbf{x} - \mathbf{x}') + \overset{\circ}{\mathbf{C}}_\mathbf{x}(\mathbf{x} - \mathbf{x}' + \Phi_x) = \vec{0} \tag{4.18c}$$

$$\mathbf{L}_x\Delta\mathbf{x} + \overset{\circ}{\mathbf{C}}_\mathbf{x}\Delta\mathbf{x} + \overset{\circ}{\mathbf{C}}_\mathbf{x}\Phi_x = \vec{0} \tag{4.18d}$$

$$(\mathbf{L}_x + \overset{\circ}{\mathbf{C}}_\mathbf{x})\Delta\mathbf{x} = -\overset{\circ}{\mathbf{C}}_\mathbf{x}\Phi_x \tag{4.18e}$$

The summation $\mathbf{L}_x + \overset{\circ}{\mathbf{C}}_\mathbf{x}$ is a symmetric, positive definite and highly sparse matrix. The multiplication $\overset{\circ}{\mathbf{C}}_\mathbf{x}\Phi_x$ produces a constant vector. This results in a system of linear equations the solution of, derives the distance the modules move, $\Delta\mathbf{x}$. Then, the new module positions are calculated as, $\mathbf{x} = \mathbf{x}' + \Delta\mathbf{x}$.

Figure 4.4 illustrates the forces exerted on a component (represented by a point) in the x-direction. At the beginning of the placement iteration the module starts from an initial position x' where the net force is equal to the hold force. Due to some overlap occurring at this position, a move force is generated that pushes the component forward towards its target point $\overset{\circ}{x}$. As the module moves, the net force increases until it neutralizes the impact of the move force and the component stops. On the start of the next iteration the hold force increases to meet the new net force to decouple from the previous step and the procedure continues as previously.

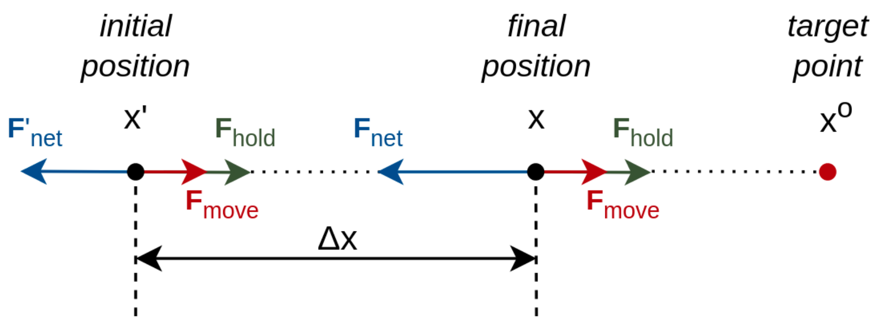


Figure 4.4: Force Equilibrium solution of one placement iteration

Algorithm 2 represents the pseudocode of Kraftwerk2. At first, an initial placement is computed by minimizing the quadratic cost function 4.1 over a few iterations. This initial placement results in a minimal netlength and high overlap solution with components concentrated mostly around the center. Then, in global placement, the modules are spread iteratively over the chip, by utilising move force and hold force. The global placement iterates, until the module overlap drops below a certain limit e.g. 20%. Most of the runtime of Kraftwerk2 is spent on solving the Poisson equation (4.13) and the system of linear equations (4.18d).

Algorithm 2: Kraftwerk2

```
// Quadratric Placement
for $i < I_{init}$ do
    // For x-direction:
    Create: $\mathbf{L}_x, \mathbf{b}_x$;
    Solve: $\mathbf{L}_x\mathbf{x} + \mathbf{b}_x = \vec{0}$;

    // For y-direction:
    Create: $\mathbf{L}_y, \mathbf{b}_y$;
    Solve: $\mathbf{L}_y\mathbf{y} + \mathbf{b}_y = \vec{0}$;

    $i++$;
end
// Global Placement
repeat
    Create: $D(x, y)$;
    Solve: $(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2})\Phi(x, y) = -D(x, y)$// Determine potential $\Phi(x, y)$;

    // determine module movement in x-direction
    Create: $\mathbf{L}_x, \mathbf{b}_x, \overset{\circ}{\mathbf{C}}_{\mathbf{x}}, \mathbf{\Phi}_x$ ;
    Solve: $\mathbf{L}_x\Delta\mathbf{x} + \overset{\circ}{\mathbf{C}}_{\mathbf{x}}\Delta\mathbf{x} + \overset{\circ}{\mathbf{C}}_{\mathbf{x}}\mathbf{\Phi}_x = \vec{0}$;
    $\mathbf{x} = \mathbf{x}' + \Delta\mathbf{x}$ // update module x-positions;

    // determine module movement in y-direction
    Create: $\mathbf{L}_y, \mathbf{b}_y, \overset{\circ}{\mathbf{C}}_{\mathbf{y}}, \mathbf{\Phi}_y$ ;
    Solve: $\mathbf{L}_y\Delta\mathbf{y} + \overset{\circ}{\mathbf{C}}_{\mathbf{y}}\Delta\mathbf{y} + \overset{\circ}{\mathbf{C}}_{\mathbf{y}}\mathbf{\Phi}_y = \vec{0}$;
    $\mathbf{y} = \mathbf{y}' + \Delta\mathbf{y}$ // update module y-positions;
until Module Overlap $\leq 20\%$;
```

4.2 Mapping Idea

As aforementioned our method works by assigning weights to critical nets to increase the net force these nets exert on components. The general mapping process is summarized in figure 4.5.

First of all, weights are bounded by an upper and lower value. As the slack worsens, the weight increases. Slacks below the average of the slack distribution are considered critical and they receive weights > 1 to increase their net force. Slacks above the average are considered non-critical and they receive weights < 1 to relax their net force.

Experimental results showed that lower weights helped into reducing the WNS and TNS metrics further. That is because relaxing the net force of non critical components, helps into their separation from critical and thus they don't contribute to the overlap of the critical areas. This means that the move force exerted on critical components is smaller and this helps into keeping them closer together. Also the weight lower bound, ensures that no zero and negative values exist.

It is also important to note, that the definition of criticality is based on the average slack reference and not on the zero point. Everything is a matter of prioritizing those that are worse of the rest and not those that simply have a negative value. In a hypothetical scenario in which the whole slack distribution happens to be positive, in timing driven placement we don't want to relax the net force with weights < 1 but to prioritize those that have worse slack than the others and aim for further improvement.

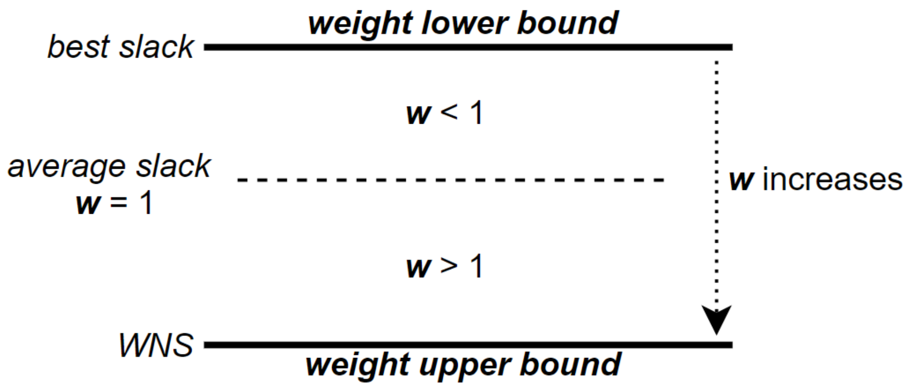


Figure 4.5: Mapping Idea

Figure 4.6 illustrates the slack histogram of a design. The x-axis corresponds to slack windows and the y-axis to the corresponding number of components. The blue color and red color represent positive and negative slack values respectively. The yellow color represents the slacks that belong close to the average value. Based on what is discussed above, the yellow color separates the histogram into two segments, one critical which receives weights > 1 and a non-critical which receives weights < 1.

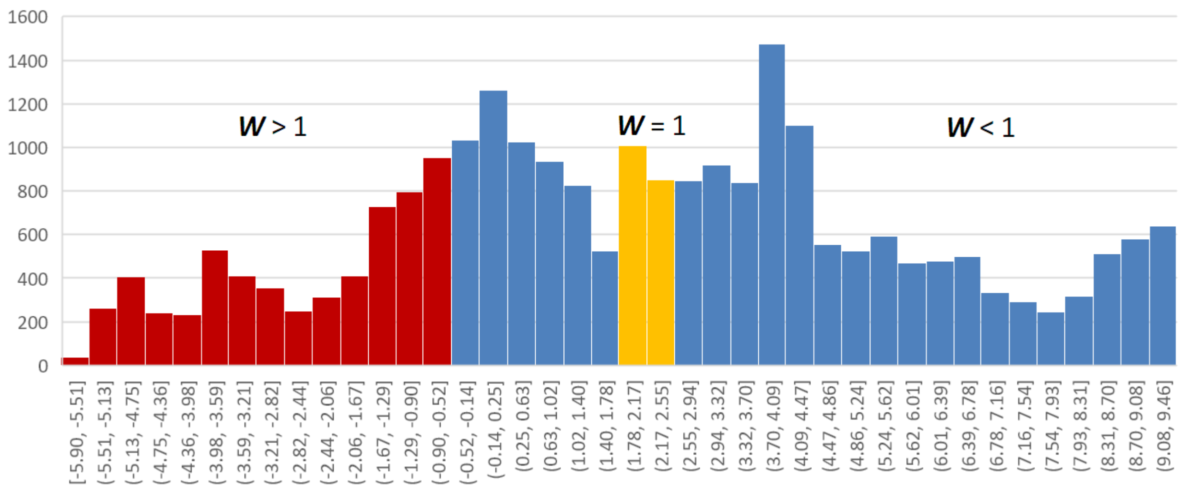


Figure 4.6: Slack Histogram

A question not answered yet, is how the specific values of the weights are generated. As we've said earlier we want the weights to increase gradually as we move to the left of the x-axis and slack values worsen. This simplest way to achieve that is by using a linear function bounded by the weight lower bound and the weight value equal to one. This function is illustrated in figure 4.7 which maps the above slack histogram into weights. Weight functions will be discussed in detail, in the following section.

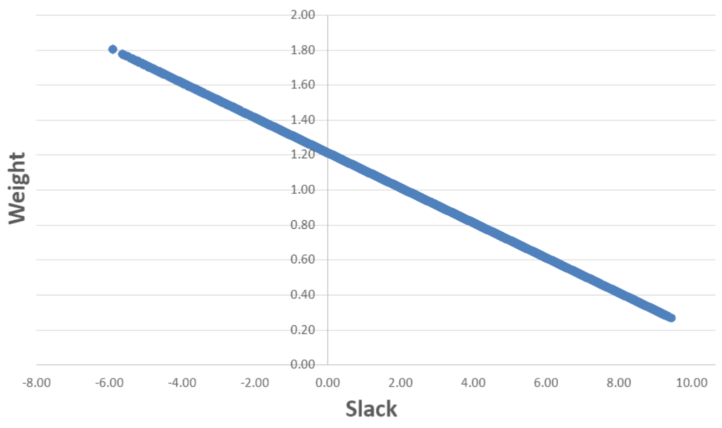


Figure 4.7: Sample Mapping Function

Here it is also the right time to discuss one main drawback of our net weighting approach. Making some targeted nets shorter during placement may sacrifice the wirelength of other nets that are connected through common cells. This tends to increase the overall wirelength of the design and also non-critical nets might become critical. Experimental results showed that the Half Perimeter Wirelength Increase (HPWL) is relative to the aggressiveness of the weight function i.e. how big are the weights it generates. There is a trade off between the increase in wirelength and the WNS and TNS drop. However if the function is too aggressive, in some designs both the wirelength and the timing metrics might worsen. This penalty must be kept generally low because although timing might improve in the placement state, subsequent states might suffer from wire congestion and routability issues leading to worse results on the finalized chip state both in terms of timing and power.

4.3 Weight Functions

4.3.1 Linear Function

The linear weight function, is defined by specifying two points and drawing the line between them. The first (blue color) is called the threshold point, and corresponds to a $weight = 1$, generated for the average of the slack distribution s_{avg}. The second point (yellow color) corresponds to the lower weight bound w_{min}, generated for the maximum slack of the design s_{max}. The third point corresponds to the maximum weight generated by the function, w_{max} by the worst slack of the design. This value, cannot be explicitly specified in this weight function and depends on its slope.

The reason we don't specify the slope of the function based on the w_{max} value instead of the w_{min}, is because we need to ensure that the latter always has a positive value.

We will now derive the formula of the linear function represented in figure 4.8.

We first start by determining the slope which is the tangent of the line connecting the two points:

$$a = \frac{1 - w_{min}}{s_{avg} - s_{max}}$$

Next, we know that any linear function can be specified by its slope and a point of choice (x_o, y_0), that lies on the function as, $y - y_0 = a(x - x_0)$. By using $(s_{avg}, 1)$ as the point of choice, we have:

$$\mathbf{W}(s) - 1 = a(s - s_{avg})$$
$$\mathbf{W}(s) = 1 + \frac{1 - w_{min}}{s_{avg} - s_{max}}(s - s_{avg})$$

Which gives the final form of the linear function:

$$\mathbf{W}(s) = 1 - \frac{1 - w_{min}}{s_{max} - s_{avg}}(s - s_{avg}) \tag{4.20}$$

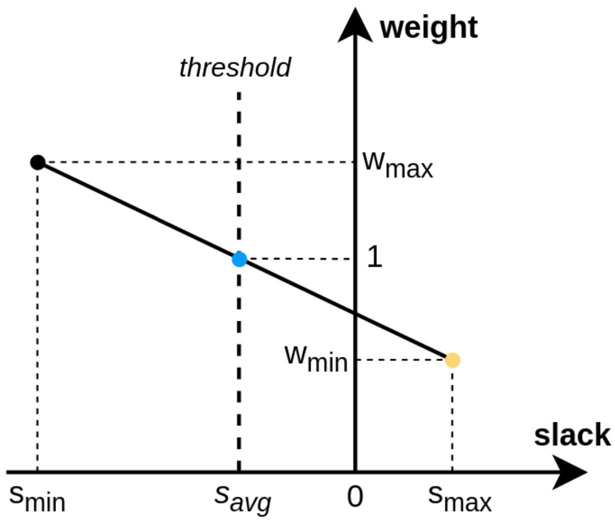


Figure 4.8: Linear Weight Function

The problem of the linear function, is the inability to control the maximum weight value w_{max}. The latter is very important because it controls the magnitude of the net-force acting on the critical components. Experiments showed that the weights generated by the linear function, proved to be small, due to the general low impact on the TNS and WNS reduction. However the linear function was the best of the rest in terms of the expense in wirelength.

4.3.2 Piece Wise Linear Function

The piecewise linear function (*PWL*) is constructed with the aim of being able to specify the w_{max} value and achieving finer control over the weight assignment of the critical nets. This function consists of two independent linear segments, joining at the threshold point. The first segment, is exactly the linear function defined above and the second segment, is a new linear function specified by the (s_{min}, w_{max}) and $(s_{avg}, 1)$ points.

The slope of the new linear segment is defined as:

$$a = \frac{w_{max} - 1}{s_{min} - s_{avg}}$$

And its formula is:

$$\begin{aligned} \mathbf{W}(s) - 1 &= a(s - s_{avg}) \\ \mathbf{W}(s) &= 1 + \frac{w_{max} - 1}{s_{min} - s_{avg}}(s - s_{avg}) \\ \mathbf{W}(s) &= 1 - \frac{w_{max} - 1}{s_{avg} - s_{min}}(s - s_{avg}) \end{aligned}$$

In conclusion, the general formula of the segmented function is:

$$\mathbf{W}_{PWL}(s) = \begin{cases} 1 - \frac{1 - w_{min}}{s_{max} - s_{avg}}(s - s_{avg}) & s_{avg} \leq s \leq s_{max} \\ 1 - \frac{w_{max} - 1}{s_{avg} - s_{min}}(s - s_{avg}) & s_{min} \leq s < s_{avg}) \end{cases} \tag{4.22}$$

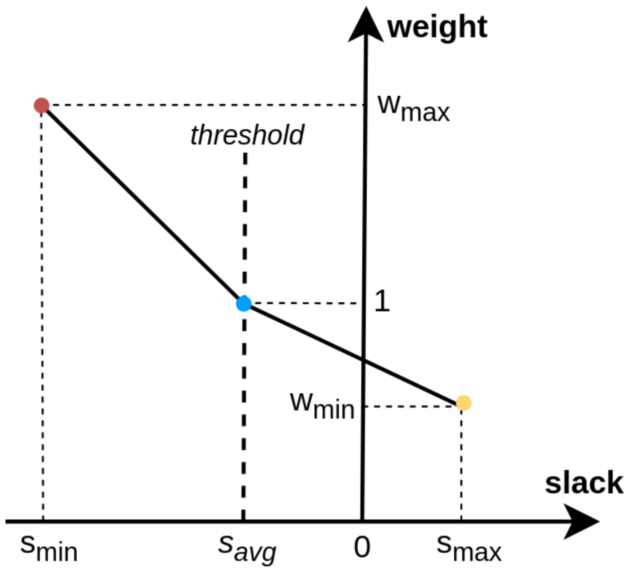


Figure 4.9: Segmented Mapping Function

By specifying both the maximum and minimum weight values, the piecewise linear function gives higher control over the net assignment process. The slope of the left segment can now be increased independently and critical nets can be prioritized more by generating higher weights. Experiments showed that this function results in substantial reduction of the TNS and WNS metrics, way above the reduction of the linear function.

4.3.3 Slow Piece Wise Exponential Function

As aformentioned, a common drawback of timing-driven placers, is the increase they induce on the total wirelength of the design. This increase is relative to the aggressiveness of the net weight assignment process. This means that using higher w_{max} values, results in a greater expense of excess wirelength.

The piecewise exponential function (*PWE*), attempts to mitigate this problem by following a softer approach till it reaches the maximum weight w_{max}, value. This is achieved by replacing the linear segment of the (*PWL*) function with an exponential segment which is bounded by the (s_{min}, w_{max}) and $(s_{avg}, 1)$ points and stays always below the linear segment. This is illustrated at figure 4.10.

The dashed line represents the corresponding left segment of the previous piecewise linear function and it is given for reference. For values less than the $(s_{min}$, the exponential segment surpasses the linear segment but of course these values do not belong to the slack domain. However this observation is important because it is exploited later for the further improvement of this function.

The formula of the (*PWE*) function, is specified as an exponential function mirrored by the y-axis and shifted s_{avg} units to the left:

$$\mathbf{W}(s) = e^{-a(s-s_{avg})} \tag{4.23}$$

Parameter a ensures that the function gives the maximum weight for the worst slack of the design, i.e. it satisfies the condition, $\mathbf{W}(s_{min}) = w_{max}$. Subtituting the value of s_{min} at function (4.23) and solving for a gives the following:

$$\begin{aligned} \mathbf{W}(s_{min}) &= w_{max} \\ e^{-(a \cdot w_{min} + s_{avg})} &= w_{max} \\ -(a \cdot w_{min} + s_{avg}) &= ln(w_{max}) \\ -a \cdot w_{min} &= ln(w_{max}) - s_{avg} \\ a &= -\frac{ln(w_{max})}{w_{min} - s_{avg}} \end{aligned}$$

and the general formula of the piecewise exponential function is:

$$\mathbf{W}_{PWE}^{slow}(s) = \begin{cases} 1 - \frac{1-w_{min}}{s_{max}-s_{avg}}(s - s_{avg}) & s_{avg} \leq s \leq s_{max} \\ e^{-\frac{ln(w_{max})}{w_{min}-s_{avg}}(s-s_{avg})} & s_{min} \leq s < s_{avg}) \end{cases} \tag{4.25}$$

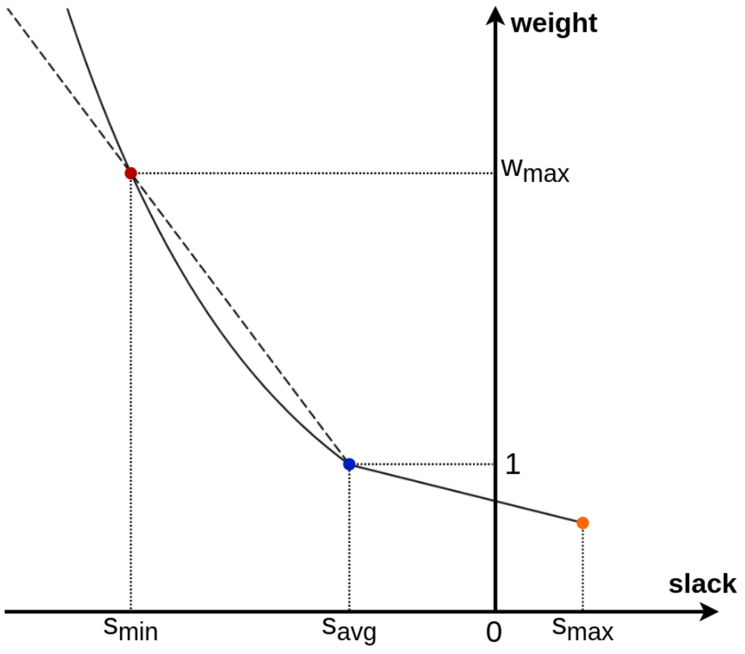


Figure 4.10: Exponential Mapping Function

From experimental results the above approach showed a decrease in the total wirelength expense relative to the piecewise linear counterpart, but the TNS and WNS drop was substantially less. In general this net-weighting approach proved to be less aggresive than required.

4.3.4 Fast Piece Wise Exponential Function

The best outcome is to achieve high WNS and TNS reduction and limit the wirelength increase at the same time. The fast piece wise exponential function attempts just that and it's illustrated in figure 4.11. It is defined by separating the left segment slack domain into two discrete zones. One critical zone which consists of very critical components that follows an aggressive net weight assignment and one non-critical zone which follows a more relaxed assignment and tries to recover from the wirelength expense.

For better visualization, the linear and slow exponential function are also represented with the dashed and bold black lines respectively. As seen, the fast exponential function is above the linear in the critical zone and below in the non-critical.

This new exponential segment is calculated based on the two points seen with the purple and blue color. The former separates the two zones and the latter separates the two segments. The purple point is also where the linear and the exponential function meet. In that way, the linear segment is used as a reference.

The critical zone is picked to be a percentage of the slack domain of the exponential function. In this example it is chosen to be the 30% of the domain. It is also important to note that upper bound weight w'_{max} is not implicitly specified and depends on the slope of the function. It is ensured however that $w'_{max} > w_{max}$ at all times.

Algorithm 3 shows how the new exponential function is derived.

Algorithm 3: Calculate fast *PWE* function

```
   // Calculate the left segment domain
1  $\Delta s = s_{min} - s_{avg}$;
   // Define critical zone as 30% of the domain
2  $critical\_zone = 0.3 \cdot \Delta s$;
   // Find the point $(c_x, c_y)$ which seperates the critical and
      non-critical zone
3  $c_x = s_{min} - critical\_zone$;
4  $c_y = \mathbf{W_{PWL}}(c_x)$;
   // Use points $(c_x, c_y)$ and $(s_{avg}, 1)$ to define the fast PWE
      function.
5  $a = \frac{ln(c_y)}{c_x - threshold}$;
6  $\mathbf{W^{fast}_{PWE}}(s) = e^{-a(s-s_{avg})}$;
```

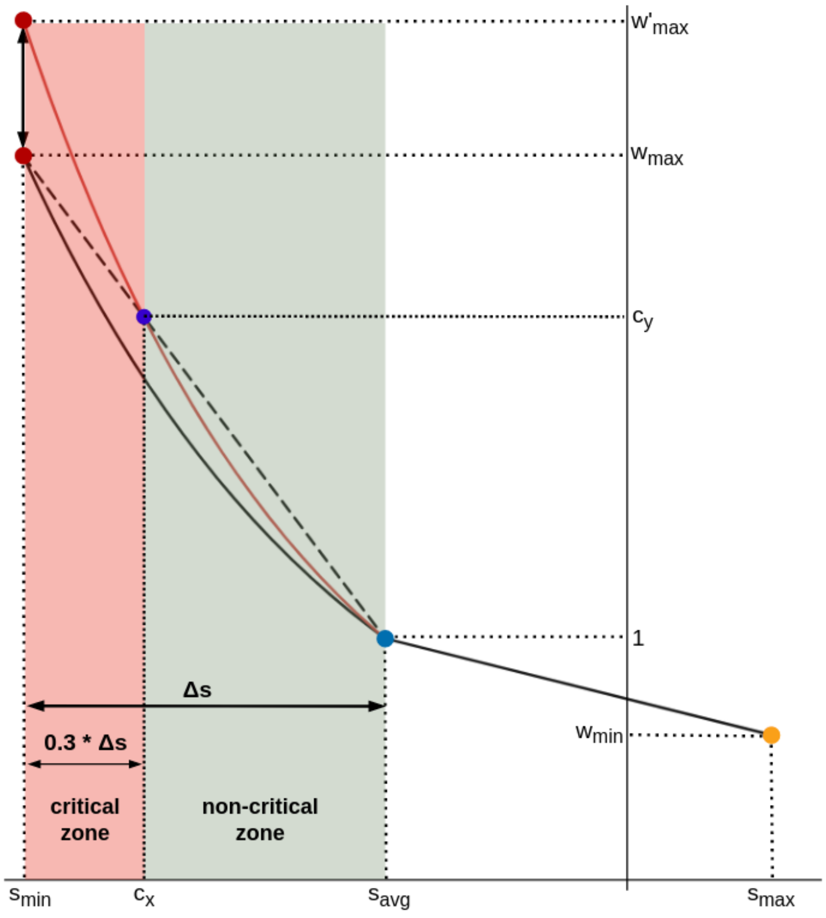


Figure 4.11: Fast Piece Wise Exponential Function

4.4 Timing Driven Kraftwerk2

4.4.1 Weight Distribution

The weight function determines a weight based on the slack of the output gatepin of a component. It then spreads this weight to the net defined by this gatepin. This approach is more pessimistic in nature and tries to minimize nets as much as possible even if an edge of a net might not be critical. This method also works in conjunction with the way the construction of the Laplacian matrix is made which is the following, Each component is traversed and the components it connects to are specified, based on its output gatepin. If a connection exists between a driver component i and a sink component j, then both of the entries $\mathbf{L}(i, j)$ and $\mathbf{L}(j, i)$ are created. In that way we can create the whole matrix by following the net definition.

However in a design, multiple output components exist. Following is an algorithm that shows how to determine a weight, on multiple output components. This is depicted in figure 4.12. A component with two output pins which defines two different nets between the input pins {3,4} and {5,6,7}. First a weight is calculated using the slack values of the output pins. Then the maximum value is determined and it is spread in both of the nets. As shown the negative slack s_2 produces a higher weight w_2 which is then spread to every edge of the two nets my multiplying the connection costs which correspond to the entries of the Laplacian matrix.

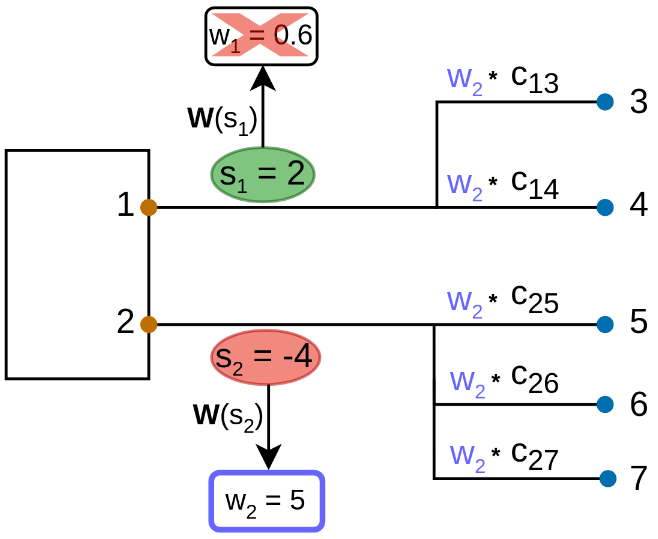


Figure 4.12: Timing Weight Net Distribution

Another question that will probably arise is, why the maximum weight is spread everywhere, even to non critical connections which will probably lead to excess HPWL? This is done for two reasons. First, because the number of multiple output components is very small and their effect can be considered negligible. Second, for an ease of implementation because in our software, when we construct the Laplacian matrix, we only check the connecting edges of the component i.e. with which components we connect to. We do not check to which net an edge belongs to i.e. from which input pin it originates from.

Finally a special case is clarified which is the occasion where some pins of a component might be unconstrained. Unconstrained means timing requirements are unspecified and the slack is infinite. This can occur in two different scenarios:

- At least one constrained pin exists (Figure 4.13).
- Constrained pin does not exist (Figure 4.14).

In the first case we simply exclude the unconstrained pin/pins from the maximum weight calculation. In the second case all the pins are excluded so a weight cannot be calculated. We solve this problem by spreading a neutral weight equal to one which leaves the connection cost values intact. That is due to the fact that unconstrained connections don't need to be prioritized by the placer and increase their net-force.

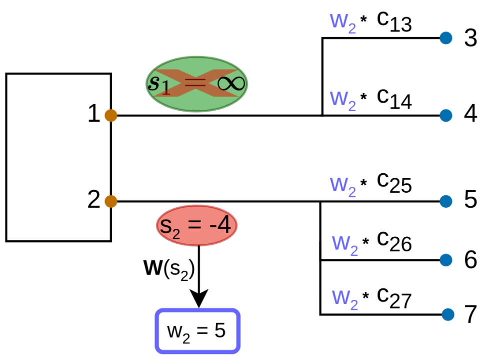


Figure 4.13: Constrained pin exists

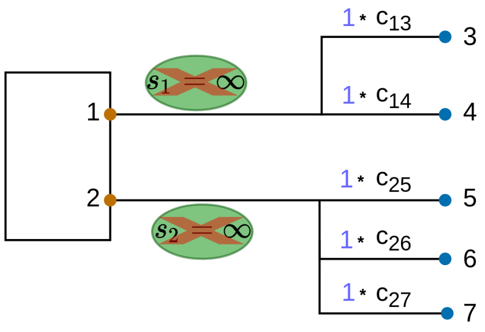


Figure 4.14: Constrained pin does not exist

Following is the algorithm that implements the above procedure written in a function called *get_component_weight*() which returns the weight corresponding to a specific component. Variable *out_pin_num* corresponds to the number of ouput pins the component has and parameter *constrained_flag* specifies if the gatepin is constrained or not. Using *weight_function*() we calculate a weight based on the slack of the output gatepin. The *weight_function*() can be any of the aforementioned mapping functions and it is user defined by a global parameter.

Algorithm 4: get_component_weight()

```
Input: component index
Output: weight
$max\_weight = -inf$;
foreach $output\_gatepin \in component$ do
 {
    $out\_pin\_num++$;
    // Exclude unconstrained pins from max calculation
    if $constrained\_flag == 1$ then
     {
        continue;
     }
    // Calculate weight
    $weight = weight\_function(slack)$;
    if $max\_weight > weight$ then
     {
        $max\_weight = weight$;
     }
 }
// If at least one constrained pin exists
if $max\_weight != -inf$ then
 {
    return $max\_weight$;
 }
// All output pins are unconstrained
else
 {
    return $1$;
 }
```

4.4.2 General Algorithm

So far we've described how the weights are generated by the various functions and how they are distributed in the network. In this chapter we describe how the previous work, is linked with the core of the global placement algorithm. This is illustrated in figure 4.15.

First, the quadratic placement runs as normally, in which the components are concentrated at the center of the chip. At this stage, no weights are inserted. It continues with the execution of the global placement algorithm.

Our work begins by running Static Timing Analysis (STA) to obtain the timing information of the circuit. STA runs always at the first placement iteration and the subsequent runs, are parameterized via specific variables. We specify both the maximum number of STA to be run as well as the interval between two successive runs. The next step, derives the essential parameters, used by the weight functions. These are the minimum, maximum slack of the design and the average of the slack distribution. In the final step, we calculate a weight for every component of the design and we distribute it on its every connection. This is done concurrently with the creation of the matrices.

An open and important question is, at which stage we should begin our timing analysis so the information we obtain is meaningful. For example shall we start from an unplaced netlist or some initial placement? Information obtained at later placement iterations, might be more accurate but at the same time, the weights generated won't have the same impact. That is because in force directed placement, components spread out at each iteration and they don't return back to old positions due to the utilization of the hold force. This means that connections between critical components cannot shorten as time goes on, by increasing their weights, but only to avoid their further increase. For this reason it is essential to generate weights from the first placement iteration and avoid the sudden spreading of critical components. A possible approach, would be to run a non-timing driven placement to obtain a layout, run a timing analysis and then use this timing information to start a timing driven placement. However due to the very large run time, this approach is not feasible.

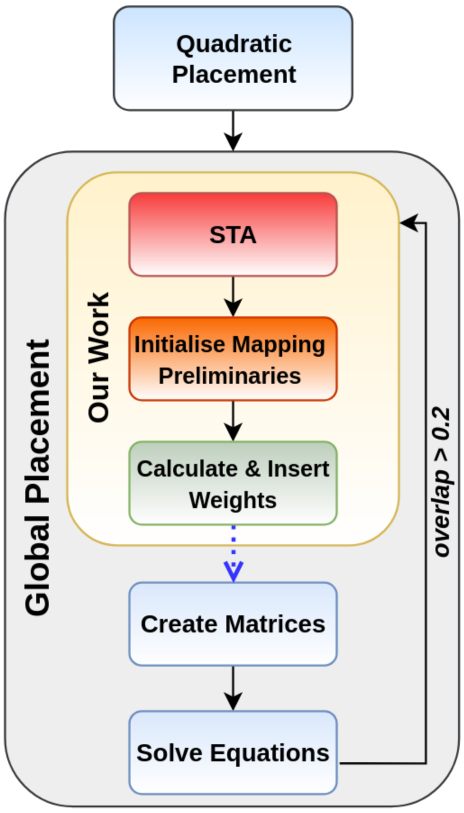


Figure 4.15: Timing Driven Global Placement

In our case, we run Static Timing Analysis immediately after Quadratic Placement. Experimental results showed that, this approach gave the best results in terms of WNS and TNS reduction instead of initiating STA at a later early stage, e.g. the 3rd or the 4th iteration. Also this approach manifests the advantage of referencing our slack to weight mapping based on the average of the slack distribution. In the post quadratic placement stage, all the components are concentrated at the core center and the only connections that contribute to the delay are those to the I/Os. For this reason the TNS value is very small compared to later more representative placement stages but due to the fact that we use the average, high weights are still generated, and critical components are identified.

Next in algorithm 5 we present the pseudocode of the Timing Driven Global Placement. It is identical to the Kraftwerk2 with the addition of two new blocks of code, one that obtains/updates timing information through static timing analysis and one where it injects the timing information into the placement matrices.

In the first code block through the *sta_iterations* and *sta_iteration_interval* variables, we control the maximum number of iterations and the interval between two successive iterations respectively.

In the second code block we traverse every single component of the design and we use the *get_component_timing_weight* function to obtain its timing weight by using the methodology described in the previous section. Then on every edge of the net defined by this component, we multiply the weight to the connection cost (e.g. B2B) and we create the entry in the corresponding matrices.

It is also important to note, that the Quadratic Placement section, remains intact.

Algorithm 5: Kraftwerk2 Timing Driven

```
    // Quadratic Placement
    // Do not include timing information in this step
 1  Create: $\mathbf{L}_x, \mathbf{b}_x$;
 2  Create: $\mathbf{L}_y, \mathbf{b}_y$;
 3  Solve: $\mathbf{L}_x\mathbf{x} + \mathbf{b}_x = \vec{0}$;
 4  Solve: $\mathbf{L}_y\mathbf{y} + \mathbf{b}_y = \vec{0}$;
    // Global Placement
 5  $i = 0$;
 6  $j = sta\_iteration\_interval$;
 7  repeat
        // Update Timing Information
 8      if $i \leq sta\_iterations$ then
 9          if $j == sta\_iteration\_interval$ then
10              Run: $Static\_Timing\_Analysis()$;
11              $i++$;
12              $j = 0$;
13          end
14          $j++$;
15      end
        // Inject Timing Information
16      foreach $component \in design$ do
17          get_component_timing_weight();
18          foreach $connection \in component$ do
19              $entry = component\_weight \cdot connection\_cost$;
20              Create $entry$ in: $\mathbf{L}_x, \mathbf{b}_x$;
21              Create $entry$ in: $\mathbf{L}_y, \mathbf{b}_y$;
22          end
            // Update module positions
23          Solve: $\mathbf{L}_x\Delta\mathbf{x} + \mathring{\mathbf{C}}_\mathbf{x}\Delta\mathbf{x} + \mathring{\mathbf{C}}_\mathbf{x}\mathbf{\Phi}_x = \vec{0}$;
24          Solve: $\mathbf{L}_y\Delta\mathbf{y} + \mathring{\mathbf{C}}_\mathbf{y}\Delta\mathbf{y} + \mathring{\mathbf{C}}_\mathbf{y}\mathbf{\Phi}_y = \vec{0}$;
25          $\mathbf{x} = \mathbf{x}' + \Delta\mathbf{x}$;
26          $\mathbf{y} = \mathbf{y}' + \Delta\mathbf{y}$ ;
27      end
28  until $Module\ Overlap \leq 20\%$;
```

Chapter 5

Experimental Results

This chapter presents the results of this thesis. Our Timing-Driven Global Placer is implemented in C language and is integrated into an existed, under development EDA tool called ASP. This tool has been used exclusively for the duration and completion of these tests. We used six benchmarks in total. The machine in which the tests run is a server with an 12-core Intel Xeon(R) CPU E5-2620 v4 @ 2.10 Ghz, physical memory of 64GB, running on Fedora 20.

5.1 Flow

In our testing, the flow of figure 5.1 was followed. First a gate-level netlist is imported into our tool in a verilog format. The gate level netlist is produced by the RTL model by maping the logic gates into real gates specified by a standard-cell library. A standard cell library is a collection of combinational and sequential logic gates that adhere to a standardized set of logical, electrical, and physical policies. In our case we use a 25nm standard-library from a company called IHP. From this library we import two files. The first one (.lef) is called a liberty file. It captures abstract information about the logical functionality, the capacitance of each input pin, leakage power and timing. The second file (.lef) includes information about the dimensions of the cells, the location and dimension of both the power/ground and signal pins and information on blockages. It can be thought as an abstract view of the GDS file produced in the finalized design state.

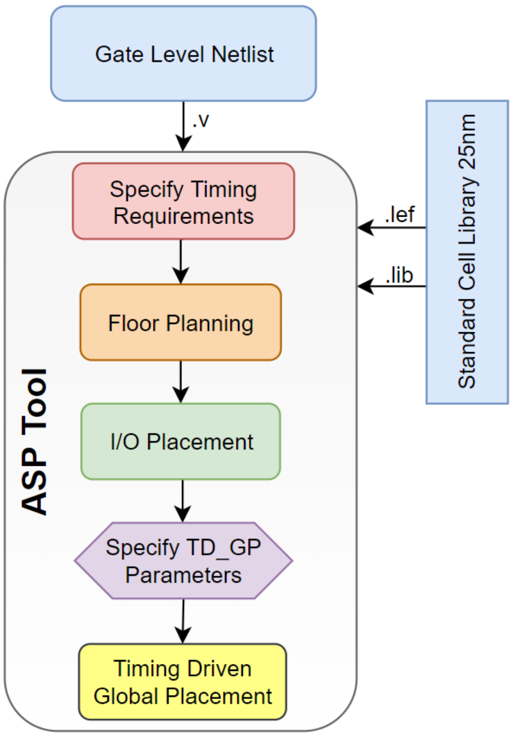


Figure 5.1: Experiments Flow

Next we specify the timing requirements of the design i.e. the clock period. Following is an early stage in the hierarchical chip design approach, which is the Floor Planning. Here, the netlist is grouped into functional blocks and these blocks are positioned in way so that they don't overlap with each other. Next is the placement of the input and output ports of the chip.

The final two stages belong to this thesis interest. In the first one we specify the global parameters that tune the behavior of the timing driven placer. Specifically we give values to the following parameters:

- *function_type*
- *sta_iteration_interval*
- *sta_iterations*
- w_{min}
- w_{max}

Parameter *function_type* determines which weight function we use for the net assignment process. In the below testcases we test every single function for every design We use a constant value for the STA parameters. That is, *sta_iteration_interval* = 2 and *sta_iterations* = 5. Then we try different combinations of { w_{min}, w_{max} }. In case of the linear mapping function, only the minimum weight is specified and the combinations are denoted as, { w_{min}, x } where '*x*' stands for "don't care" value. Next the timing driven global placement begins and runs iteratively until the overlap metric objective is met.

The different weight combinations { w_{min}, w_{max} } are specified in the following manner:

1. Test *Linear* weight function for different w_{min} values starting from value 0.1 and ending at value 0.9 at increments of 0.1 .
2. Use the w_{min} value that produces the biggest reduction in the WNS and TNS metrics, as the lower bound for the other weight functions.
3. Experiment with different w_{max} values.

For every type of function, we present graphs in Apendix A that illustrate the impact of the weight assignment on the design metrics. The graph with the dark blue color ({x,x}) represents the value of a conventional non timing-driven placement. Green and red color represent a reduction/increase of any amount, respectively. We also present the percentages of decrease/increase in metric values relative the non-td placement, in tables. Finally a table illustrates the best results obtained for each weight function.

The infinite number of weight pair combinations and the great variety of mapping functions resulted in a high number of testcases. This in conjuction with the limited amount of computing resources, required us to test designs with a relative small number of components due to the very large CPU time overhead. The purpose was to test the behavior of the net-weight assignment and experiment with as many values as possible in a reasonable amount of time. By testing small designs we can identify a range of optimal weight values that work well and use them for larger designs with the hope that the timing metrics reduction will scale well. The designs tested are summarized in table 5.1.

Benchmarks	**# Cells**
`PID`	3401
`AES`	7487
`AES192`	9195
`apbAES192`	10201
`LDPC`	52506
`b19`	218976

Table 5.1: OpenCores Benchmarks Characteristics

5.2 Results

PID

The results of the Linear mapping function are presented in figure 5.2 and table 5.2. It can be seen that the smaller the w_{min} values are, the bigger the reduction in the WNS and TNS metrics is. That is mainly true for two reasons.

First, choosing small w_{min} values increases the gradient of the linear function and achieve higher weights in the critical slack area.

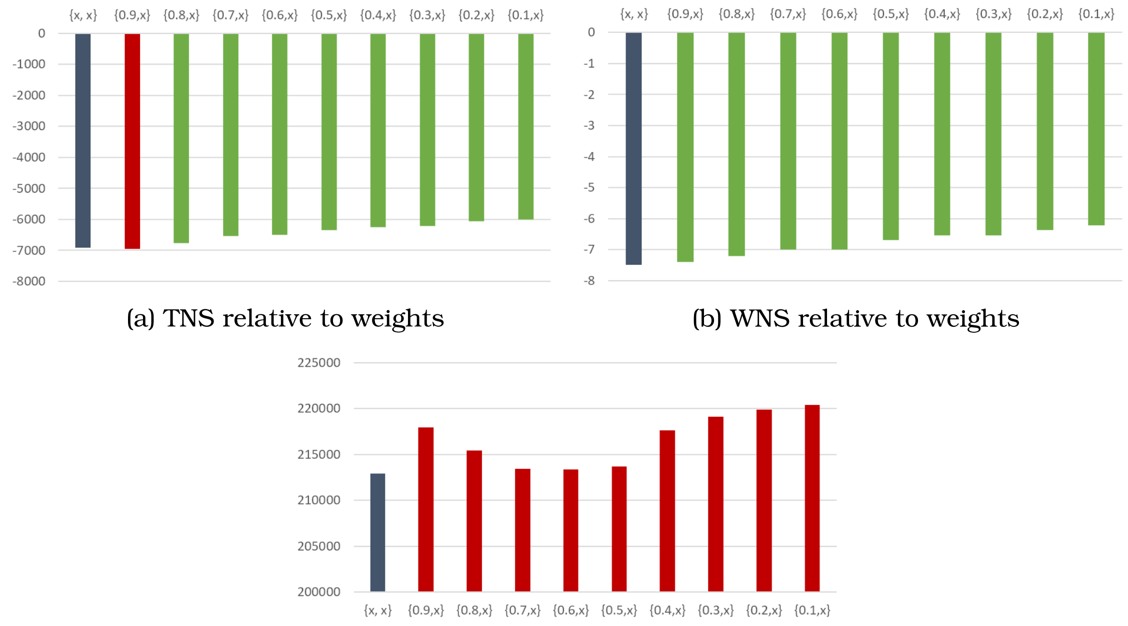
Second, lower weights relax the net force of non critical components and increase their move distance. In that way, non-critical components move away from critical and separate. In that way, they do not interfere with the density calculation of critical areas which is responsible for the generation of the move force. If non-critical components lie above critical, they might increase the move force of the area and critical components will be pushed further while the goal is to keep them as close as possible.

It is also clear that the HPWL increase is relative to the reduction of the timing metrics. That is because critical components concentrate closer together and their connections through common cells increase.

Next in figure 5.3 and table 5.3 we present the results for a piece wise linear (*PWL*) and a slow piece wise exponential (PWE_{slow}) weight function. Here the generation of higher weights achieves substantially better reduction on the timing metrics but with a higher expense in HPWL. Comparing *PWL* and PWE_{slow} we can see that PWE_{slow} has smaller increase in HPWL for almost every weight bound pair. That is because this function follows a less aggressive net weighting approach by always being below the corresponding left segment of the *PWL* function.

Finally in figure 5.4 and table 5.8 we present the results of the most aggressive fast piece wise exponential (PWE_{fast}) weight function. This function is below the left segment of the *PWL* function for the non-critical zone and above for the critical zone, achieving values higher than w_{max}. We remind that w_{max} does not represent the maximum weight the function generates but the maximum value the *PWL* would generate if it was used instead. We provide two different variations of this function determined for two different critical zone (*CZ*) values. We can see that the 50%

critical zone decreases the timing metrics sooner but it has higher expense in HPWL and the end result considering the timing metrics is the same.



(a) TNS relative to weights

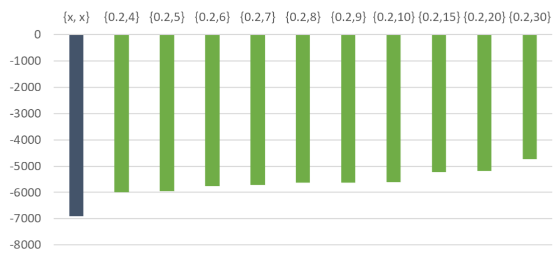
(b) WNS relative to weights

(c) HPWL relative to weights

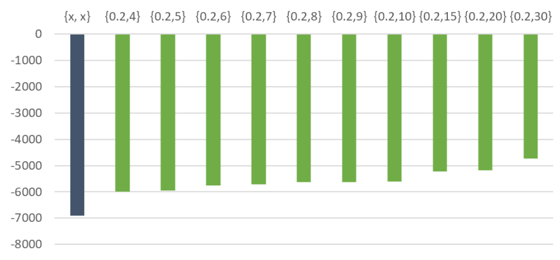
Figure 5.2: Impact of *Linear* weight function on design metrics

Weight Bounds	HPWL	WNS	TNS
{0.9,x}	+2.3%	-1.3%	+0.5%
{0.8,x}	+1.2%	-3.9%	-2.1%
{0.7,x}	+0.2%	-6.5%	-5.6%
{0.6,x}	+0.2%	-6.5%	-6.0%
{0.5,x}	+0.4%	-10.5%	-8.2%
{0.4,x}	+2.2%	-12.8%	-9.5%
{0.3,x}	+2.9%	-12.7%	-10.1%
{0.2,x}	+3.3%	-15.0%	-12.5%
{0.1,x}	+3.5%	-17.1%	-13.3%

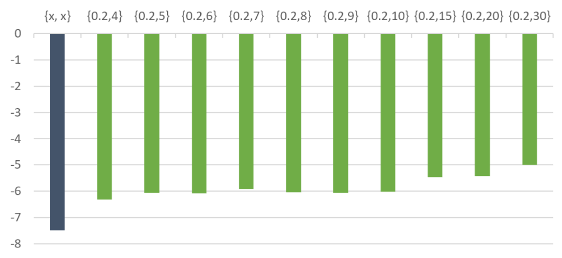
Table 5.2: Percentage change relative to conventional placement for *Linear* weight function



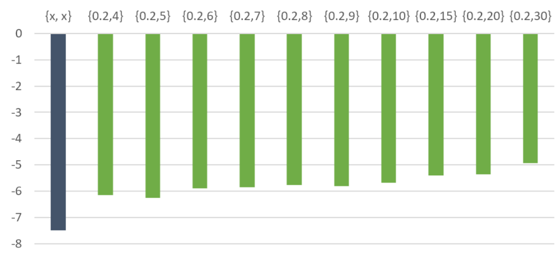
(a) TNS relative to weights for *PWL*



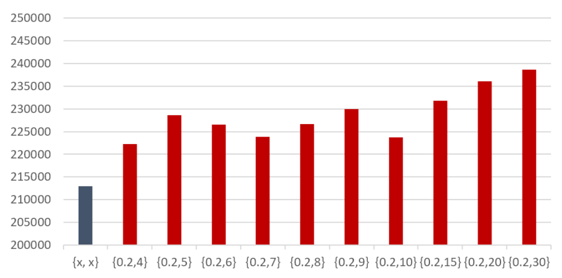
(b) TNS relative to weights for PWE_{slow}



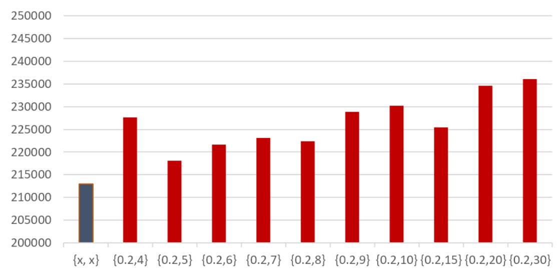
(c) WNS relative to weights for *PWL*



(d) WNS relative to weights for PWE_{slow}



(e) HPWL relative to weights for *PWL*

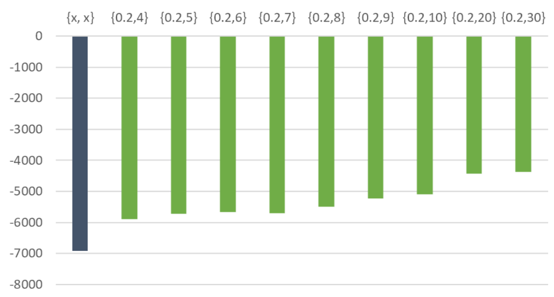


(f) HPWL relative to weights for PWE_{slow}

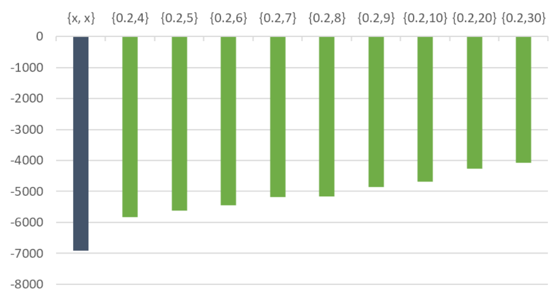
Figure 5.3: Impact of *PWL* & PWE_{slow} weight functions on design metrics

	PWL			PWE_{slow}		
Weight Bounds	HPWL	WNS	TNS	HPWL	WNS	TNS
{0.2,4}	4.4%	-15.8%	-13.1%	6.9%	-18.0%	-13.4%
{0.2,5}	7.4%	-19.0%	-16.1%	2.4%	-16.4%	-13.9%
{0.2,6}	6.4%	-18.7%	-15.9%	4.1%	-21.2%	-16.9%
{0.2,7}	5.1%	-21.0%	-17.1%	4.8%	-21.9%	-17.4%
{0.2,8}	6.4%	-19.2%	-17.0%	4.4%	-23.1%	-18.5%
{0.2,9}	8.0%	-19.0%	-18.3%	7.5%	-22.4%	-18.6%
{0.2,10}	5.1%	-19.8%	-17.7%	8.1%	-24.3%	-19.0%
{0.2,15}	8.8%	-27.1%	-24.0%	5.8%	-27.8%	-24.4%
{0.2,20}	10.8%	-27.6%	-25.8%	10.2%	-28.4%	-25.2%
{0.2,30}	12.1%	-33.2%	-31.7%	10.9%	-34.0%	-31.6%

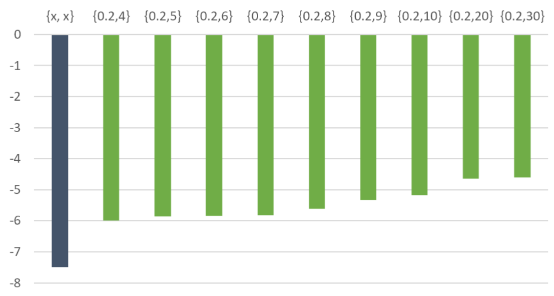
Table 5.3: Percentage of change relative to conventional placement for *PWL* & PWE_{slow} weight functions



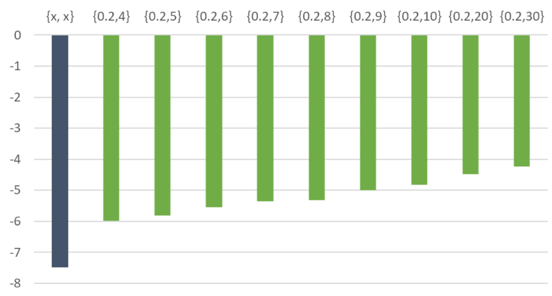
(a) TNS relative to weights for *CZ* = 30%



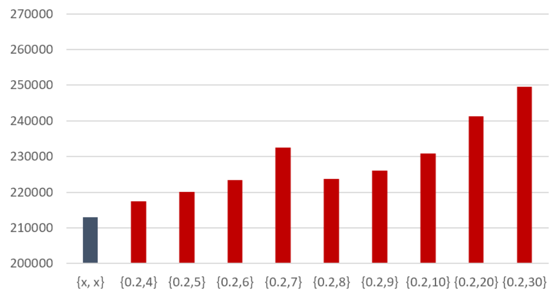
(b) TNS relative to weights for *CZ* = 50%



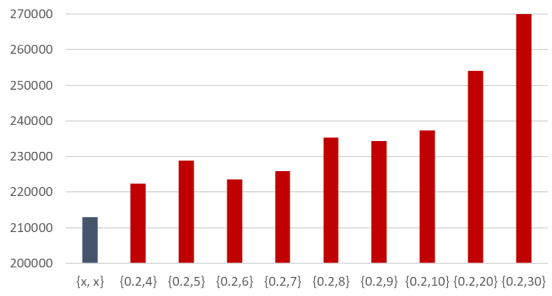
(c) WNS relative to weights for *CZ* = 30%



(d) WNS relative to weights for *CZ* = 50%



(e) HPWL relative to weights for *CZ* = 30%



(f) HPWL relative to weights for *CZ* = 50%

Figure 5.4: Impact of PWE_{fast} weight funtion on design metrics

	Critical Zone = 30%			Critical Zone = 50%		
Weight Bounds	HPWL	WNS	TNS	HPWL	WNS	TNS
{0.2,4}	6.0%	-21.5%	-17.2%	8.1%	-21.5%	-16.1%
{0.2,5}	8.3%	-21.9%	-17.1%	7.5%	-22.0%	-17.8%
{0.2,6}	6.2%	-24.0%	-18.2%	6.8%	-28.0%	-23.2%
{0.2,7}	7.2%	-24.8%	-24.5%	8.8%	-27.9%	-25.2%
{0.2,8}	7.2%	-29.9%	-26.0%	9.5%	-30.8%	-27.7%
{0.2,9}	8.3%	-31.1%	-27.6%	12.2%	-32.3%	-28.8%
{0.2,10}	8.4%	-32.8%	-28.5%	13.4%	-36.2%	-33.0%
{0.2,20}	14.4%	-38.6%	-38.0%	21.5%	-40.7%	-39.2%
{0.2,30}	19.4%	-41.9%	-39.9%	28.9%	-42.2%	-40.6%

Table 5.4: Percentage of change relative to conventional placement for PWE_{fast} weight function

Mapping Function	HPWL	WNS	TNS
Linear	+3.3%	-17%	-13.3%
PWL	+12%	-33%	-32%
PWE$_{slow}$	+11%	-34%	-32%
PWE$_{fast}$	+19.4%	-42%	-40%

Table 5.5: Best results per weight function

For the following designs, the graphs which represent the values of the wire-length and timing metrics, are depicted in the Appendix A.

AES

We will now proceed with the results of the AES design. Here we follow the exact same flow as above by first presenting the results of the *Linear* weight function. The results are similar to the PID design which has the characteristic of having very low HPWL expense.

Next the results of the *PWL* & *PWE$_{slow}$* are illustrated. Here the latter is proven to be less aggressive than it should and cannot reach the reduction values of the former. However the HPWL increase is substantially less.

Finally we compare the results of two different *PWE$_{fast}$* functions. Here the reduction can reach values as high as 80%. However the HPWL expense is large. This is proven to be a trade-off and it must be handled differently based on the design specifications.

Weight Bounds	HPWL	WNS	TNS
{0.9,x}	+0.3%	+1.5%	-1.0%
{0.8,x}	+0.4%	-1.3%	-4.7%
{0.7,x}	+0.8%	-4.9%	-6.8%
{0.6,x}	+0.7%	-5.7%	-8.5%
{0.5,x}	+1.3%	-1.8%	-10.7%
{0.4,x}	+1.7%	0.8%	-12.1%
{0.3,x}	+2.5%	-3.9%	-14.8%
{0.2,x}	+3.8%	-11.9%	-19.0%
{0.1,x}	+6.5%	-12.1%	-21.5%

Table 5.6: Percentage of change relative to a conventional placement for *Linear* weight function

	PWL			*PWE*$_{slow}$		
Weight Bounds	HPWL	WNS	TNS	HPWL	WNS	TNS
{0.2,7}	4.6%	-22.2%	-31.3%	3.5%	-16.0%	-25.1%
{0.2,8}	4.9%	-21.3%	-32.4%	3.5%	-16.5%	-25.5%
{0.2,9}	5.1%	-25.0%	-34.4%	3.5%	-17.5%	-25.9%
{0.2,10}	5.4%	-22.7%	-33.5%	3.4%	-18.0%	-26.4%
{0.2,15}	6.7%	-19.3%	-37.8%	3.6%	-15.1%	-23.1%
{0.2,20}	8.3%	-19.3%	-42.0%	3.8%	-22.7%	-29.9%
{0.2,30}	11.3%	-23.2%	-48.3%	4.3%	-21.2%	-32.3%
{0.2,50}	13.9%	-27.4%	-58.7%	4.8%	-24.0%	-34.8%
{0.2,100}	21.3%	-38.8%	-71.5%	6.2%	-28.2%	-38.2%
{0.2,200}	-	-	-	7.4%	-29.9%	-41.2%

Table 5.7: Percentage of change relative to conventional placement for *PWL* & *PWE*$_{slow}$ weight functions

	Critical Zone = 40%			Critical Zone = 70%		
Weight Bounds	HPWL	WNS	TNS	HPWL	WNS	TNS
{0.2,20}	5.3%	-24.0%	-35.2%	9.0%	-37.3%	-49.0%
{0.2,30}	6.6%	-29.7%	-38.9%	11.4%	-43.9%	-58.0%
{0.2,50}	7.9%	-32.3%	-44.2%	16.2%	-50.1%	-67.2%
{0.2,100}	10.1%	-41.2%	-52.7%	27.7%	-52.1%	-73.9%
{0.2,150}	11.3%	-43.4%	-57.2%	34.1%	-61.0%	-78.8%
{0.2,170}	11.9%	-45.2%	-58.5%	35.7%	-61.8%	-79.2%
{0.2,200}	12.8%	-45.0%	-60.1%	37.7%	-58.7%	-80%

Table 5.8: Percentage of change relative to conventional placement for *PWE*$_{fast}$ weight function

Mapping Function	HPWL	WNS	TNS
Linear	+6.5%	-12.1%	-21.5%
PWL	+21.3%	-38.8%	-71.5%
PWE$_{slow}$	+7.4%	-29.9%	-41.2%
PWE$_{fast}$	+37.7%	-58.7%	-80%

Table 5.9: Best results per weight function

AES192

In the AES192 design the linear function has again similar results to the above. The comparison between the *PWL* & *PWE*$_{slow}$ shows that the higher aggressiveness of the former has the upper hand again, in terms of WNS and TNS reduction. Finally by presenting the results of the *PWE*$_{fast}$ weight function, we can see that it can reach TNS reduction close to 90% and for a relative small HPWL expense. This shows that AES192 design handles high weight assignment very well.

Weight Bounds	HPWL	WNS	TNS
{0.9,x}	-0.1%	+8.3%	+0.1%
{0.8,x}	-0.2%	-0.3%	-5.9%
{0.7,x}	-0.1%	-0.3%	-7.9%
{0.6,x}	+0.1%	-8.3%	-14.6%
{0.5,x}	+0.3%	-9.2%	-17.6%
{0.4,x}	+1.0%	-7.3%	-16.7%
{0.3,x}	+1.9%	-5.3%	-14.2%
{0.2,x}	+3.2%	-8.9%	-15.5%
{0.1,x}	+6.0%	-4.3%	-12.8%

Table 5.10: Percentage of change relative to conventional placement for *Linear* weight function

	PWL			*PWE*$_{slow}$		
Weight Bounds	HPWL	WNS	TNS	HPWL	WNS	TNS
{0.5,10}	+3.3%	-5.6%	-24.6%	+0.8%	-5.9%	-19.2%
{0.5,20}	+7.4%	-5.6%	-35.0%	+1.8%	-5.3%	-18.1%
{0.5,30}	+11.7%	-12.5%	-39.6%	+2.4%	-6.3%	-19.6%
{0.5,50}	+17.3%	-23.8%	-44.7%	+4.0%	-5.6%	-19.8%
{0.5,70}	+20.7%	-23.8%	-53.5%	-	-	-
{0.5,100}	+25.1%	-40.3%	-54.8%	+5.0%	-8.6%	-24.3%
{0.5,200}	-	-	-	5.8%	-16.5%	-31.1%
{0.5,300}	-	-	-	6.9%	-17.8%	-33.4%

Table 5.11: Percentage of change relative to conventional placement for *PWL* & *PWE*$_{slow}$ weight functions

	Critical Zone = 40%			Critical Zone = 70%		
Weight Bounds	HPWL	WNS	TNS	HPWL	WNS	TNS
{0.5,10}	1.9%	-6.3%	-19.2%	4.6%	-8.3%	-21.8%
{0.5,20}	4.4%	-6.9%	-20.5%	7.8%	-25.4%	-41.7%
{0.5,30}	5.2%	-13.2%	-27.2%	10.6%	-38.0%	-53.4%
{0.5,50}	7.0%	-17.8%	-33.7%	14.0%	-26.1%	-65.8%
{0.5,100}	9.2%	-28.4%	-43.9%	17.2%	-28.4%	-79.8%
{0.5,200}	11.2%	-42.2%	-58.6%	21.2%	-25.4%	-86.1%
{0.5,300}	13.3%	-27.4%	-63.0%	23.4%	-25.7%	-89.4%

Table 5.12: Percentage of change relative to conventional placement for *PWE*$_{fast}$ weight function

Mapping Function	HPWL	WNS	TNS
Linear	+6.0%	-4.3%	-12.8%
PWL	+25.1%	-40.3%	-54.8%
PWE$_{slow}$	+6.9%	-17.8%	-33.4%
PWE$_{fast}$	+23.4%	-25.7%	-89.4%

Table 5.13: Best results per weight function

apbAES128

The apbAES128 has the exact opposite behavior, relative to the previous designs. Aggressive net weight assignment, over-constraints the design and creates new critical paths. The *Linear* weight function produces the best result here.

Weight Bounds	HPWL	WNS	TNS
{0.9,x}	-7.0%	+0.3%	-13.6%
{0.8,x}	-0.9%	-21.4%	-16.4%
{0.7,x}	-3.8%	+13.1%	-3.3%
{0.6,x}	+3.3%	-14.6%	-14.1%
{0.5,x}	-0.5%	-18.5%	-14.4%
{0.4,x}	+2.5%	-20.6%	-24.9%
{0.3,x}	+4.1%	-22.6%	-31.1%
{0.2,x}	+13.4%	-19.6%	-29.6%

Table 5.14: Percentage of change relative to conventional placement for *Linear* weight function

	PWL			*PWE_{slow}*		
Weight Bounds	HPWL	WNS	TNS	HPWL	WNS	TNS
{0.3,4}	+22.8%	-13.9%	-20.9%	+19.9%	-17.6%	-25.2%
{0.3,5}	+19.2%	-23.2%	-28.1%	+20.6%	-28.4%	-29.9%
{0.3,6}	+20.8%	-33.2%	-28.5%	+27.1%	-33.0%	-30.6%
{0.3,7}	+28.8%	-34.5%	-28.7%	+28.1%	-36.2%	-28.5%
{0.3,8}	+29.8%	-32.6%	-23.5%	+28.9%	-33.7%	-27.1%
{0.3,9}	+31.2%	-35.8%	-29.8%	+29.5%	-36.4%	-32.2%
{0.3,10}	+41.2%	-39.6%	-13.7%	+37.6%	-34.6%	-26.3%
{0.3,20}	+57.7%	-46.4%	-31.8%	+53.8%	-42.7%	-34.2%
{0.3,30}	+79.1%	-30.3%	+43.5%	+57.4%	-39.3%	+28.6%
{0.3,50}	+98.2%	-41.0%	+31.5%	+94.8%	-41.7%	+38.3%
{0.3,100}	+108.9%	-35.0%	+40.8%	+98.1%	-36.8%	+36.8%

Table 5.15: Percentage of change relative to conventional placement for *PWL* & *PWE_{slow}* weight functions

	Critical Zone = 30%			Critical Zone = 50%		
Weight Bounds	HPWL	WNS	TNS	HPWL	WNS	TNS
{0.3,4}	+19.9%	-24.7%	-28.4%	+22.1%	-30.7%	-28.0%
{0.3,5}	+23.6%	-33.6%	-30.8%	+29.5%	-36.4%	-32.2%
{0.3,6}	+29.8%	-34.7%	-26.4%	+29.8%	-35.1%	-26.4%
{0.3,7}	+40.2%	-38.0%	-25.3%	+39.5%	-33.4%	-26.5%
{0.3,8}	+38.8%	-22.9%	-23.1%	+47.8%	-41.4%	-28.7%
{0.3,9}	+45.6%	-28.3%	-19.1%	+59.5%	-39.2%	+17.8%
{0.3,10}	+41.5%	-36.9%	-22.3%	+71.1%	-39.1%	+36.3%
{0.3,20}	+90.6%	-42.6%	+38.0%	+106.7%	-33.9%	+46.0%
{0.3,30}	+90.2%	-43.0%	+26.9%	+102.8%	-41.4%	+21.5%
{0.3,50}	+110.8%	-32.7%	+56.3%	+141.1%	-19.1%	+99.3%
{0.3,100}	+122.5%	+13.4%	+91.5%	+184.2%	-25.8%	+91.5%

Table 5.16: Percentage of change relative to conventional placement for *PWE_{fast}* weight function

Mapping Function	HPWL	WNS	TNS
Linear	+13.4%	-19.6%	-29.6%
PWL	+20.8%	-33.2%	-28.5%
PWE_{slow}	+20.6%	-28.4%	-30.0%
PWE_{fast}	+23.6%	-33.6%	-30.8%

Table 5.17: Best results per weight function

LDPC

The LDPC design has similar behavior as the apbAES design with lower weight combinations producing better results. The TNS reduction degrades with the increase of the upper weight bounds.

Weight Bounds	HPWL	WNS	TNS
{0.8,x}	0.2%	-9.4%	-9.0%
{0.7,x}	0.5%	-10.2%	-12.3%
{0.6,x}	0.8%	-7.6%	-15.7%
{0.5,x}	1.1%	-7.4%	-19.3%
{0.4,x}	1.5%	-10.1%	-22.1%
{0.3,x}	2.1%	-20.7%	-25.0%
{0.2,x}	2.7%	-20.5%	-28.6%
{0.1,x}	3.7%	-25.3%	-33.1%

Table 5.18: Percentage of change relative to conventional placement for *Linear* weight function

	PWL			*PWE_{slow}*		
Weight Bounds	HPWL	WNS	TNS	HPWL	WNS	TNS
{0.1,5}	6.3%	-36.5%	-42.2%	6.2%	-36.0%	-42.1%
{0.1,6}	6.9%	-38.2%	-43.3%	6.8%	-37.4%	-43.0%
{0.1,7}	7.4%	-36.7%	-43.4%	7.3%	-37.9%	-43.1%
{0.1,8}	7.9%	-39.1%	-43.5%	7.7%	-37.8%	-43.4%
{0.1,9}	8.3%	-39.3%	-43.3%	8.1%	-39.3%	-42.9%
{0.1,10}	8.7%	-40.8%	-43.0%	8.4%	-39.8%	-42.7%
{0.1,20}	11.2%	-38.4%	-41.2%	10.6%	-38.4%	-40.0%
{0.1,30}	12.9%	-39.4%	-39.0%	12.0%	-37.8%	-36.9%
{0.1,40}	13.9%	-34.6%	-37.9%	12.6%	-34.4%	-36.0%
{0.1,50}	14.7%	-33.3%	-36.5%	13.2%	-38.1%	-34.3%
{0.1,100}	16.6%	-31.4%	-33.4%	14.6%	-34.9%	-32.1%

Table 5.19: Percentage of change relative to conventional placement for *PWL* & *PWE_{slow}* weight functions

	Critical Zone = 30%			Critical Zone = 50%		
Weight Bounds	HPWL	WNS	TNS	HPWL	WNS	TNS
{0.1,5}	+9.7%	-37.8%	-40.8%	+10.2%	-38.9%	-40.8%
{0.1,6}	+10.6%	-39.7%	-41.1%	+11.4%	-37.2%	-40.1%
{0.1,7}	+11.5%	-38.1%	-40.2%	+12.4%	-37.6%	-38.5%
{0.1,8}	+12.2%	-40.1%	-39.0%	+14.0%	-37.2%	-38.2%
{0.1,9}	+12.9%	-38.6%	-39.1%	+13.3%	-38.8%	-38.8%
{0.1,10}	+13.4%	-37.3%	-38.9%	+14.6%	-37.9%	-38.6%
{0.1,20}	+17.2%	-37.1%	-36.3%	+18.0%	-39.2%	-34.5%
{0.1,30}	+18.4%	-38.6%	-35.0%	+20.2%	-39.8%	-35.2%
{0.1,40}	+19.7%	-41.6%	-35.1%	+22.5%	-30.8%	-31.6%
{0.1,50}	+20.8%	-38.0%	-34.3%	+24.6%	-33.9%	-25.1%
{0.1,100}	+29.2%	-3.4%	-16.8%	+37.8%	+17.4%	+14.2%

Table 5.20: Percentage of change relative to conventional placement for *PWE_{fast}* weight function

Mapping Function	HPWL	WNS	TNS
Linear	+3.7%	-25.3%	-33.1%
PWL	+6.9%	-38.2%	-43.3%
PWE$_{slow}$	+8.1%	-39.3%	-43.0%
PWE$_{fast}$	+10.2%	-38.9%	-40.8%

Table 5.21: Best results per weight function

B19

The B19 has a peculiar behavior so far, with low w_{max} values increasing the TNS metric. As weights get more aggressive, timing metrics start to reduce.

Weight Bounds	HPWL	WNS	TNS
{0.8,x}	+3.5%	-19.0%	-9.7%
{0.7,x}	+6.8%	-15.0%	0.7%
{0.6,x}	+1.9%	-21.2%	-19.8%
{0.5,x}	+5.0%	-19.2%	-7.4%
{0.3,x}	+7.5%	-21.2%	-15.6%
{0.2,x}	+6.1%	-11.2%	-12.3%
{0.1,x}	+11.2%	-7.9%	-4.4%

Table 5.22: Percentage of change relative to a conventional placement for *Linear* mapping function

	PWL			*PWE$_{slow}$*		
Weight Bounds	HPWL	WNS	TNS	HPWL	WNS	TNS
{0.6,4}	9.0%	-19.4%	0.2%	7.6%	-21.7%	2.3%
{0.6,5}	9.9%	-19.1%	0.4%	10.3%	-15.3%	2.2%
{0.6,6}	4.8%	-22.6%	-12.6%	9.4%	-16.6%	6.7%
{0.6,7}	10.9%	-21.3%	-1.1%	4.3%	-22.1%	-7.6%
{0.6,8}	11.4%	-18.5%	-1.1%	8.9%	-13.4%	7.2%
{0.6,9}	5.2%	-22.9%	-17.6%	6.8%	-19.3%	-0.5%
{0.6,10}	12.6%	-20.2%	-12.4%	7.7%	-20.2%	-3.9%
{0.6,20}	17.8%	-20.2%	-12.7%	10.9%	-30.7%	5.7%
{0.6,30}	19.4%	-27.5%	-25.2%	12.6%	-30.7%	-1.6%
{0.6,50}	21.5%	-35.9%	-24.2%	11.1%	-38.0%	-11.0%
{0.6,100}	30.3%	-35.8%	-33.5%	17.0%	-26.6%	-16.9%

Table 5.23: Percentage of change for *PWL* & *PWE$_{slow}$* relative to a conventional placement

Weight Bounds	HPWL	WNS	TNS
{0.6,5}	+9.3%	-20.2%	+0.3%
{0.6,6}	+8.5%	-20.0%	+5.8%
{0.6,7}	+8.4%	-19.0%	+5.0%
{0.6,8}	+10.5%	-26.1%	+1.8%
{0.6,9}	+10.9%	-30.4%	+5.3%
{0.6,10}	+10.0%	-31.1%	+3.0%
{0.6,20}	+17.2%	-26.3%	-19.0%
{0.6,30}	+20.3%	-40.2%	-21.0%
{0.6,40}	+19.8%	-38.2%	-25.4%
{0.6,50}	+24.2%	-38.5%	-18.4%
{0.6,100}	+34.4%	-41.3%	-12.6%

Table 5.24: Percentage of change relative to a conventional placement for PWE_{fast} weight function

Mapping Function	HPWL	WNS	TNS
Linear	+1.9%	-21.2%	-19.8%
PWL	+30.3%	-35.8%	-33.5%
PWE_{slow}	+17%	-26.6%	-16.9%
PWE_{fast}	+19.8%	-38.2%	-25.4%

Table 5.25: Best results per weight function

Summarized Results

The following table, summarizes the best results so far in terms of TNS reduction for each design.

Design	HPWL	WNS	TNS
PID	+19.4%	-42.0%	-40.0%
AES	+37.7%	-58.7%	-80%
*AES*192	+23.4%	-25.7%	-89.4%
*apbAES*128	+23.6%	-33.6%	-30.8%
LDPC	+6.9%	-38.2%	-43.3%
*B*19	+30.3%	-35.8%	-33.5%
Average	+23.55%	-39.0%	-52.8%

Table 5.26: Best results per design

Impact of Legalization

Although the results so far have been very promising, they are not totally representative of the reality because the placement procedure is not finalized till the completion of the Legalization step. The following table shows the impact of legalization on the design metrics. It can be seen that the wirelength and WNS have a negligible increase due to the displacement of the cells. However the designs which produced the best decrease in TNS value (AES, AES192) have a big increase of the same, after legalization. The final TNS value however is still way lower than the non-timing driven placement with a 72.8% and a 60% decrease for the AES and AES192 designs respectively

Increase after Legalization			
Designs	HPWL	WNS	TNS
PID	3%	3%	3%
AES	2%	12%	33%
AES192	3%	4%	277%
apbAES182	3%	1%	2%
LDPC	1%	0%	2%
b19	2%	3%	4%

Chapter 6

Conclusions and Future Work

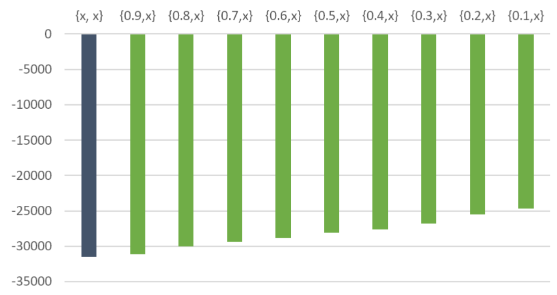
In this thesis we provided an implementation of a timing driven global placer based on a net weight assignment method. The end results proven to be very promising with substantial reduction in the WNS and TNS metrics and a relative small degradation of the total wirelength. Also apart from the calls to the STA engine, there was negligible impact in the execution time of the placer. However one caveat which is clear from the demonstration of the experimental results, is the unpredictability of this method considering the impact of weight pair values on the design metrics and the difficulty of identifying optimal values. That unpredictability is a problem, especially on large designs which require high CPU run time and make the method of trying multiple weight pair combinations unfeasible.

Further improvements must focus on ways of making the impact of weights more predictable by guessing proper weight values pre-placement. This could be done by obtaining more data for more designs and by trying to detect similarities and peculiarities between them. Machine learning could also be of great help. Another possible positive improvement would be the incorporation of sensitivity approaches as mentioned in section 3.2. In that way the net weight assignment process is not totally heuristic and a better understanding of weight impact is given, not only on timing metrics but also on other design characteristics.

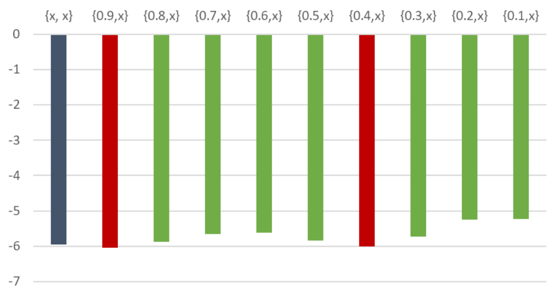
Appendix A

Result Graphs

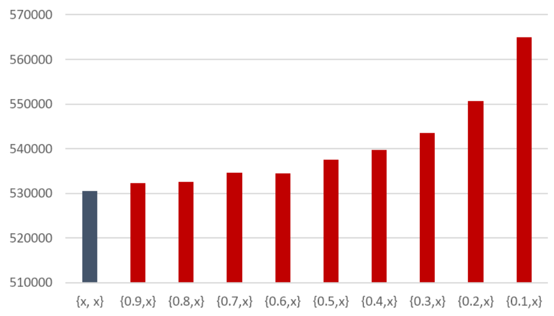
A.1 AES



(a) TNS relative to weights

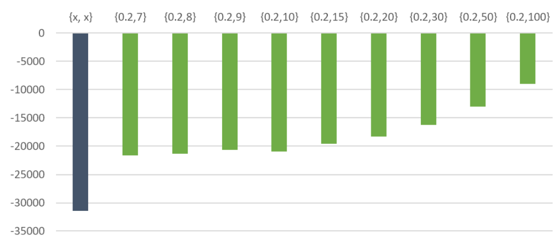


(b) WNS relative to weights

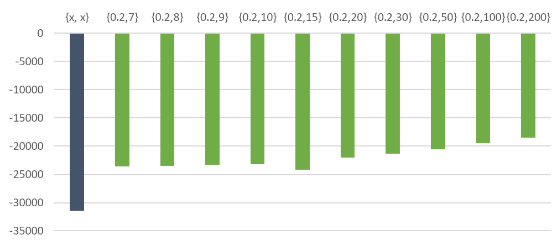


(c) HPWL relative to weights

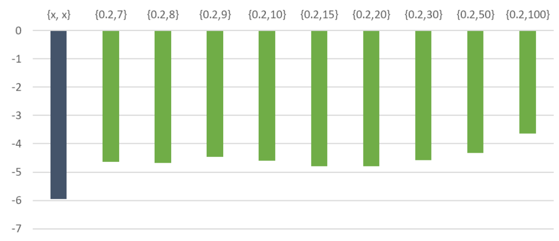
Figure A.1: Impact of *Linear* weight function on design metrics



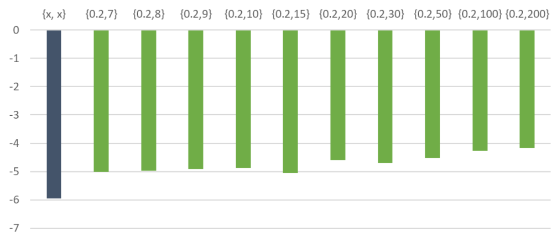
(a) TNS relative to weights for *PWL*



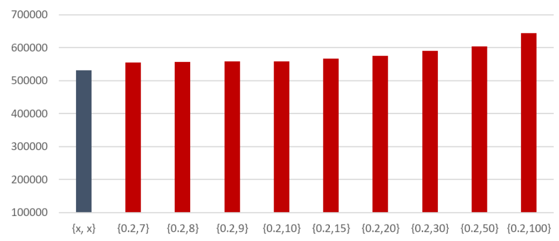
(b) TNS relative to weights for PWE_{slow}



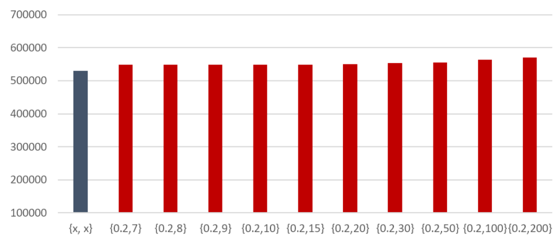
(c) WNS relative to weights for *PWL*



(d) WNS relative to weights for PWE_{slow}

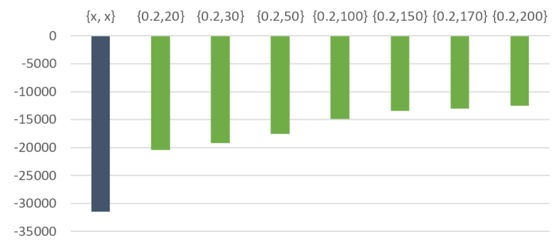


(e) HPWL relative to weights for *PWL*

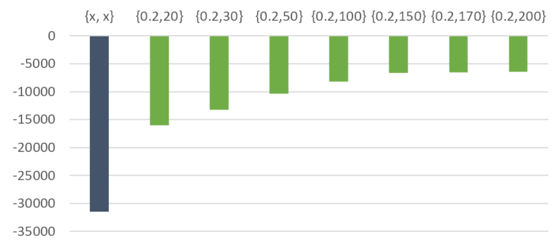


(f) HPWL relative to weights for PWE_{slow}

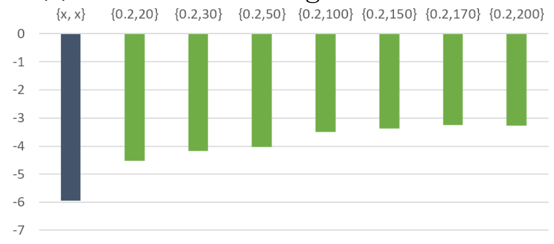
Figure A.2: Impact of *PWL* & PWE_{slow} weight functions on design metrics



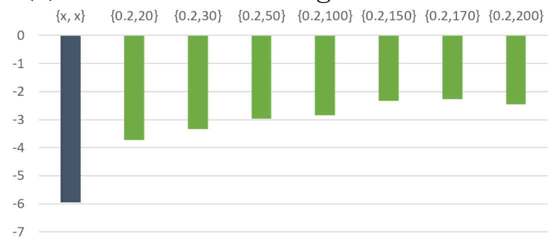
(a) TNS relative to weights for CZ = 40%



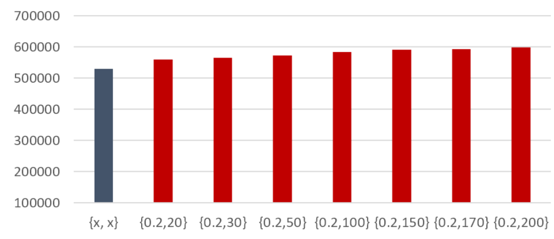
(b) HPWL relative to weights for CZ = 70%



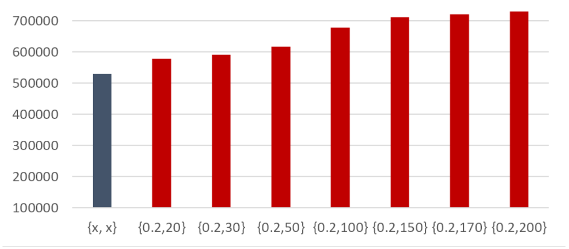
(c) WNS relative to weights for CZ = 40%



(d) WNS relative to weights for CZ = 70%



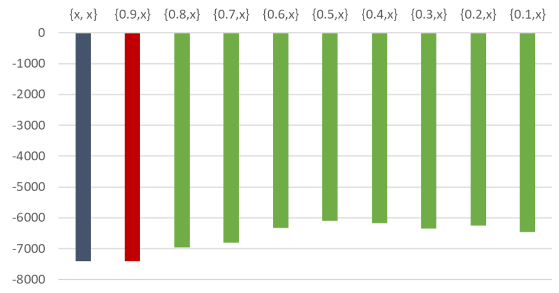
(e) HPWL relative to weights for CZ = 40%



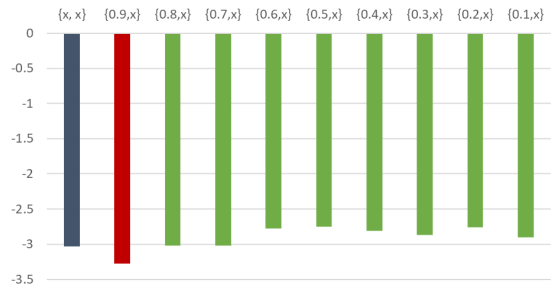
(f) HPWL relative to weights for CZ = 70%

Figure A.3: Impact of PWE_{fast} weight funtion on design metrics

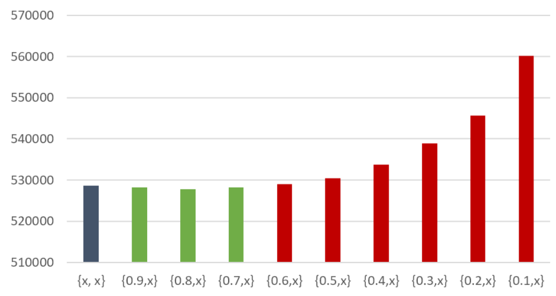
A.2 AES192



(a) TNS relative to weights

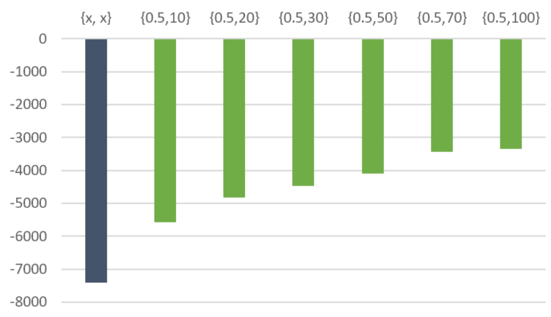


(b) WNS relative to weights

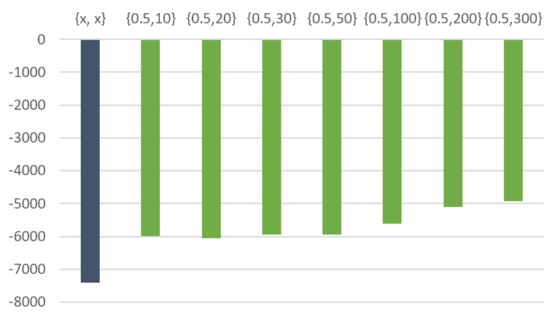


(c) HPWL relative to weights

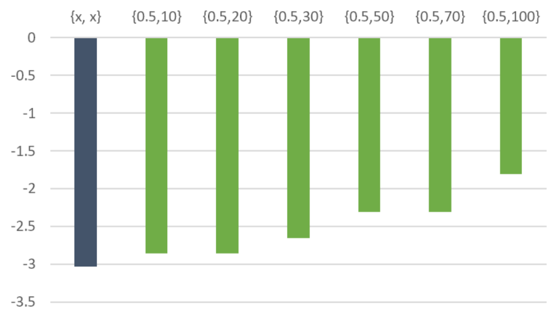
Figure A.4: Impact of *Linear* function on design metrics



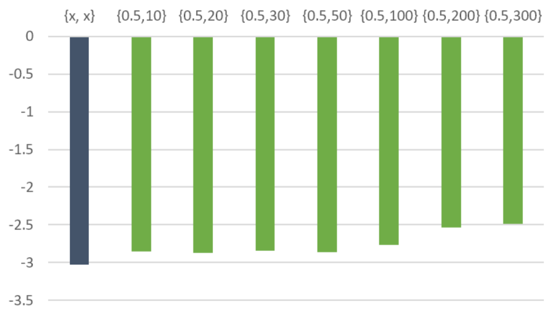
(a) TNS relative to weights for *PWL*



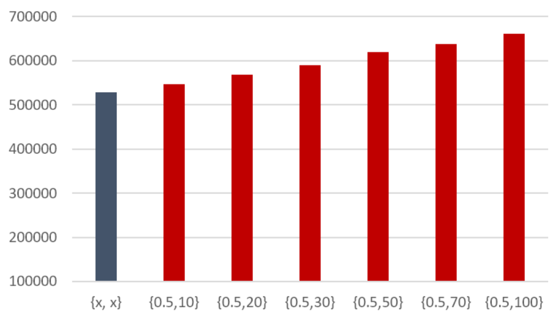
(b) TNS relative to weights for PWE_{slow}



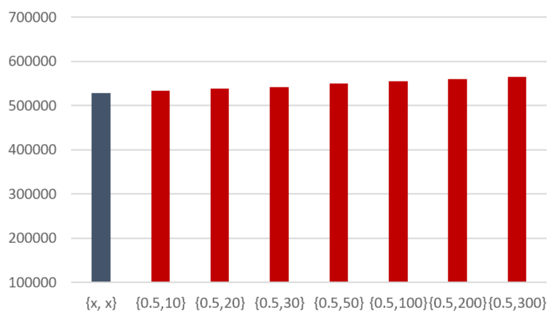
(c) WNS relative to weights for *PWL*



(d) WNS relative to weights for PWE_{slow}

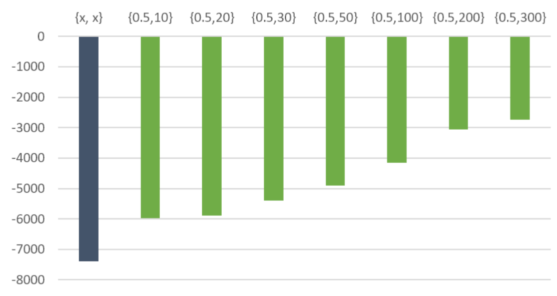


(e) HPWL relative to weights for *PWL*

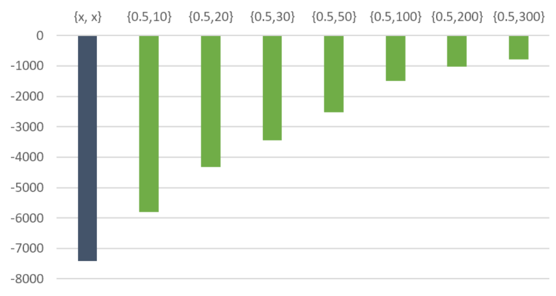


(f) HPWL relative to weights for PWE_{slow}

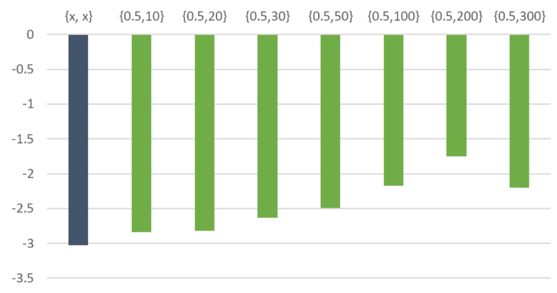
Figure A.5: Impact of *PWL* & PWE_{slow} weight functions on design metrics



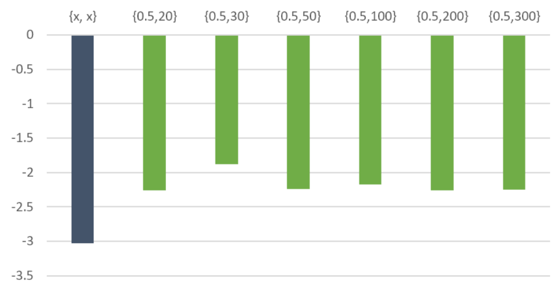
(a) TNS relative to weights for CZ = 40%



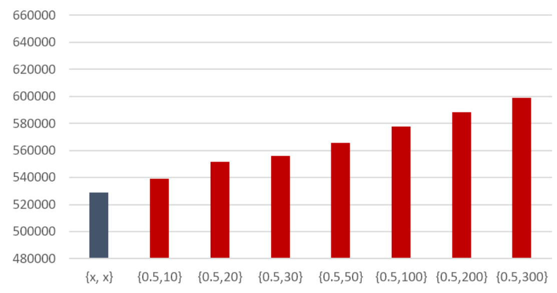
(b) HPWL relative to weights for CZ = 70%



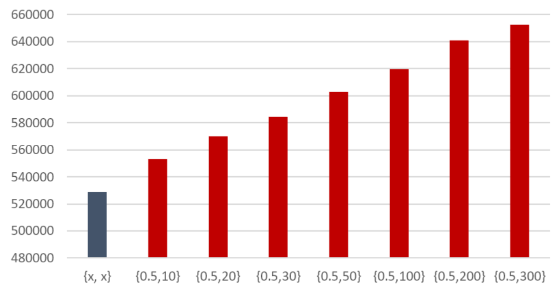
(c) WNS relative to weights for CZ = 40%



(d) WNS relative to weights for CZ = 70%



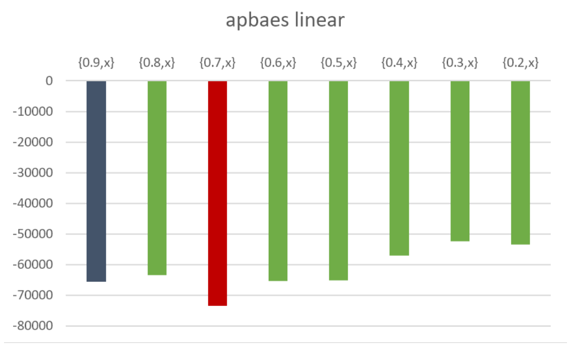
(e) HPWL relative to weights for CZ = 40%



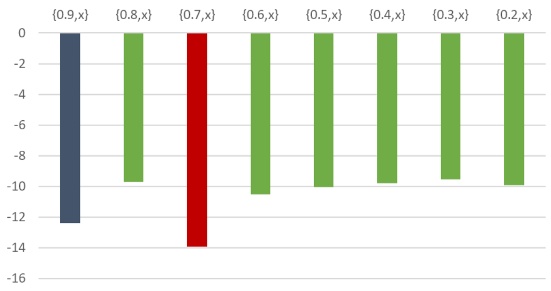
(f) HPWL relative to weights for CZ = 70%

Figure A.6: Impact of PWE_{fast} weight funtion on design metrics

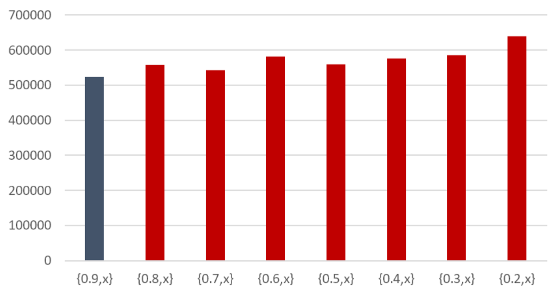
A.3 apbAES128



(a) TNS relative to weights

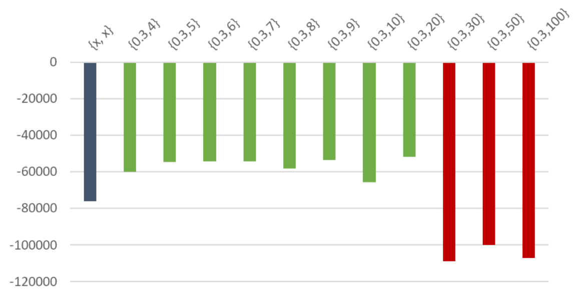


(b) WNS relative to weights

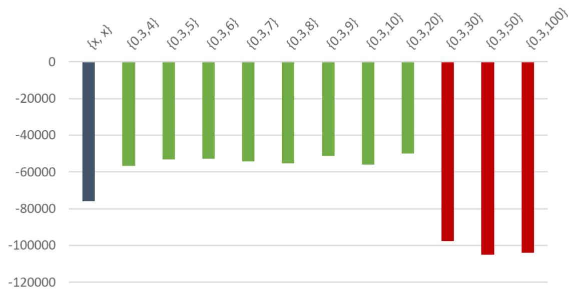


(c) HPWL relative to weights

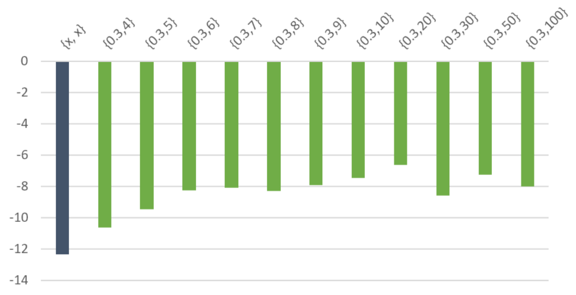
Figure A.7: Impact of *Linear* function on design metrics



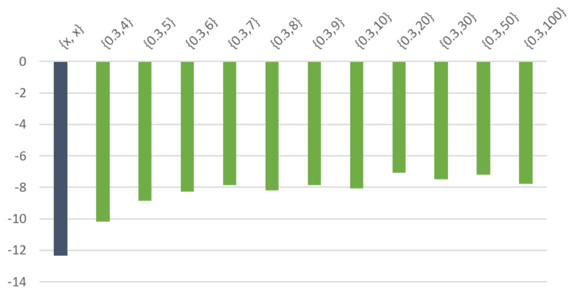
(a) TNS relative to weights for *PWL*



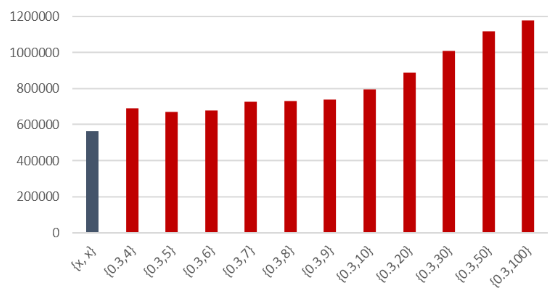
(b) TNS relative to weights for PWE_{slow}



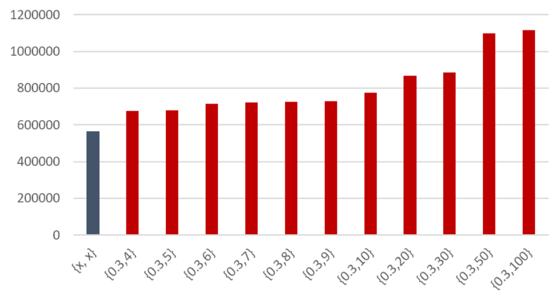
(c) WNS relative to weights for *PWL*



(d) WNS relative to weights for PWE_{slow}

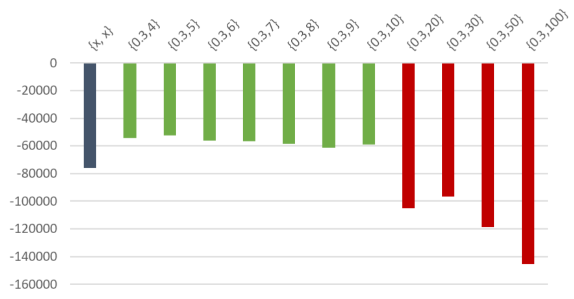


(e) HPWL relative to weights for *PWL*

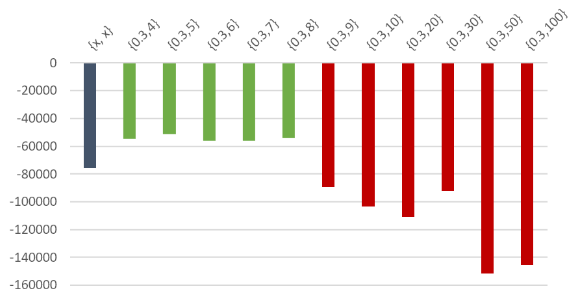


(f) HPWL relative to weights for PWE_{slow}

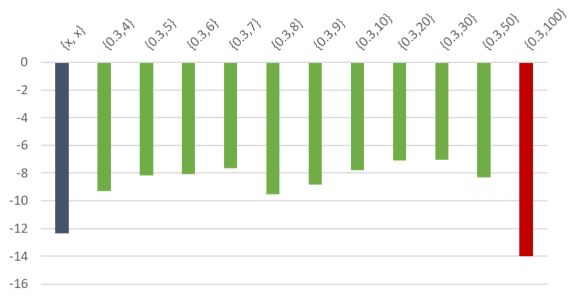
Figure A.8: Impact of *PWL* & PWE_{slow} weight functions on design metrics



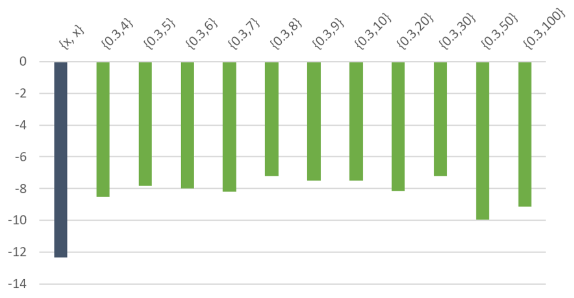
(a) TNS relative to weights for CZ = 30%



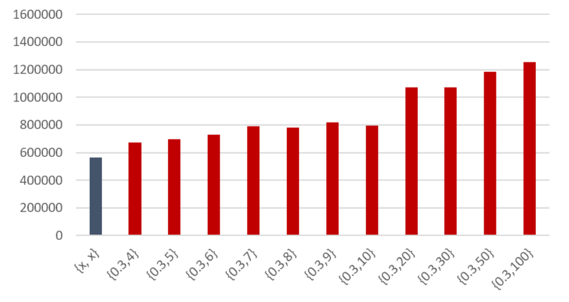
(b) HPWL relative to weights for CZ = 50%



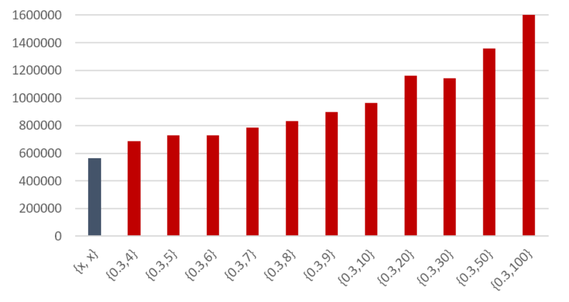
(c) WNS relative to weights for CZ = 30%



(d) WNS relative to weights for CZ = 50%



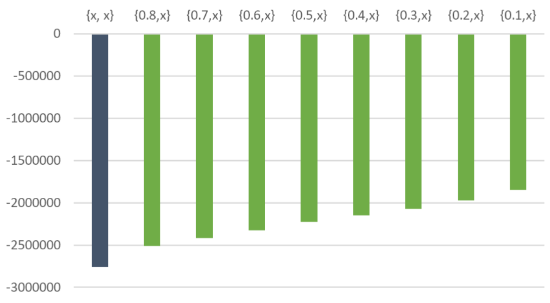
(e) HPWL relative to weights for CZ = 30%



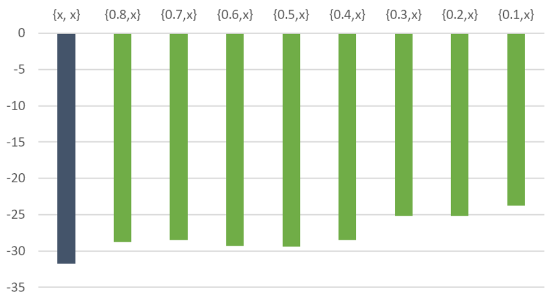
(f) HPWL relative to weights for CZ = 50%

Figure A.9: Impact of PWE_{fast} weight funtion on design metrics

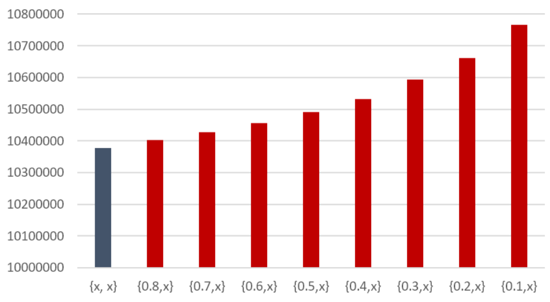
A.4 LDPC



(a) TNS relative to weights

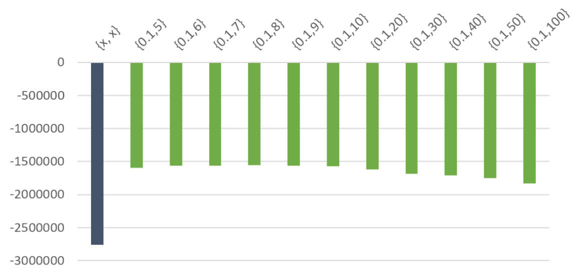


(b) WNS relative to weights

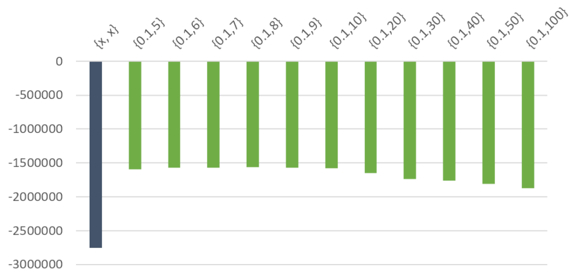


(c) HPWL relative to weights

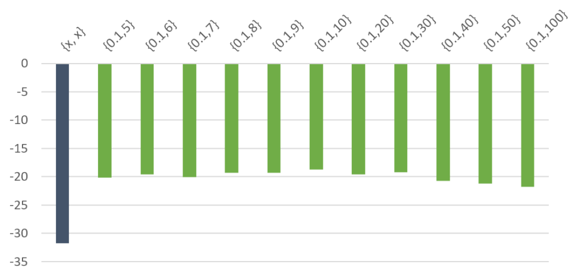
Figure A.10: Impact of *Linear* function on design metrics



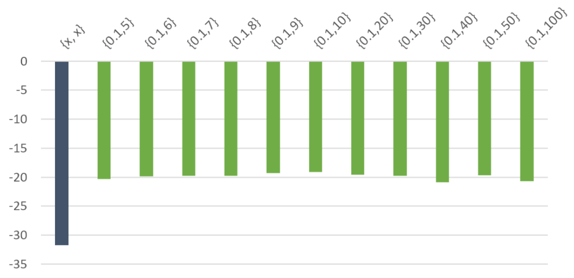
(a) TNS relative to weights for *PWL*



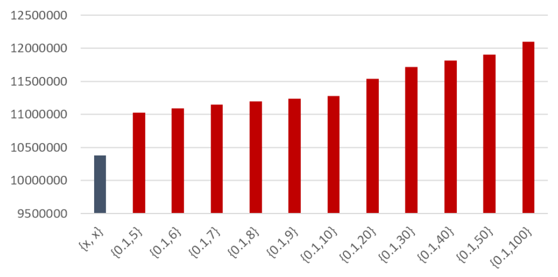
(b) TNS relative to weights for PWE_{slow}



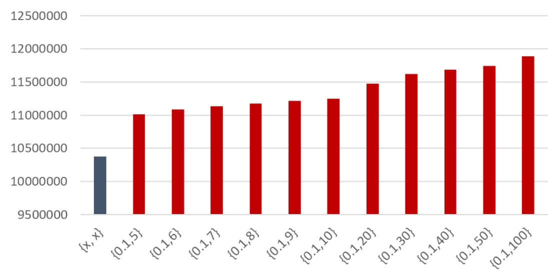
(c) WNS relative to weights for *PWL*



(d) WNS relative to weights for PWE_{slow}

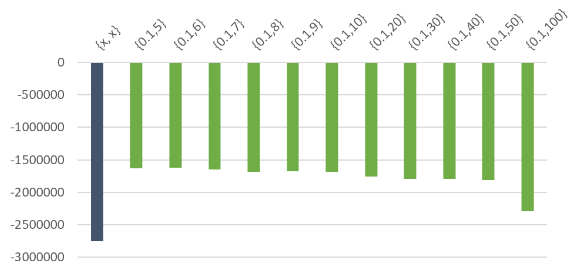


(e) HPWL relative to weights for *PWL*

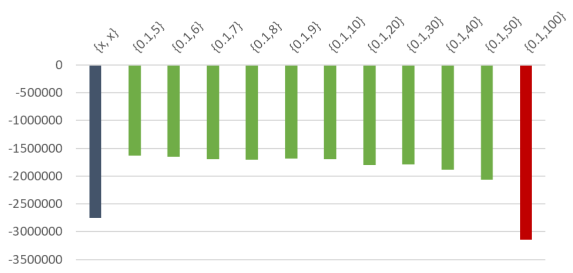


(f) HPWL relative to weights for PWE_{slow}

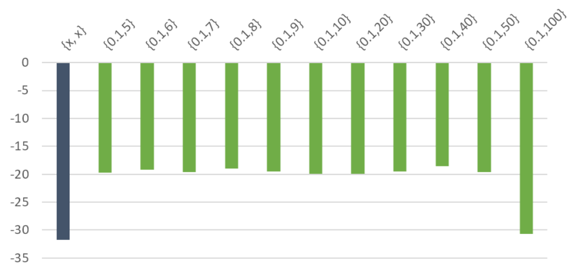
Figure A.11: Impact of *PWL* & PWE_{slow} weight functions on design metrics



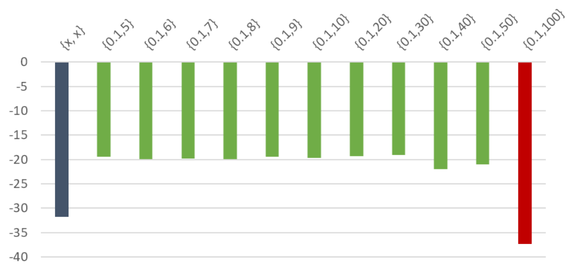
(a) TNS to weights for CZ = 30%



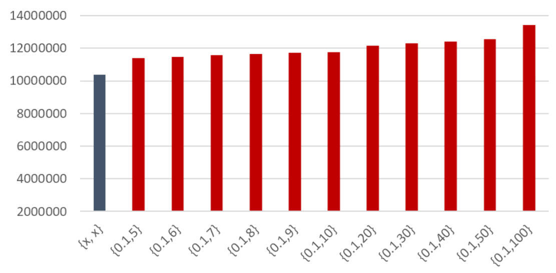
(b) HPWL to weights for CZ = 50%



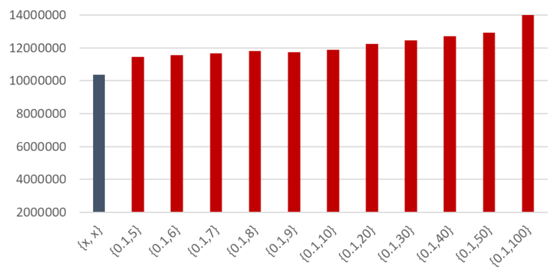
(c) WNS to weights for CZ = 30%



(d) WNS to weights for CZ = 50%



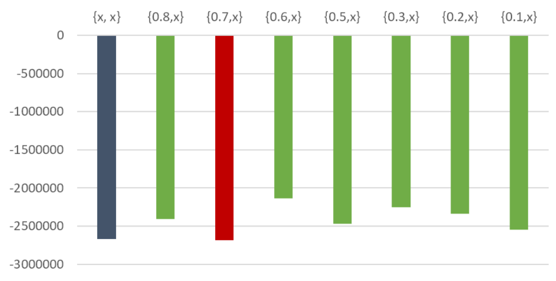
(e) HPWL to weights for CZ = 30%



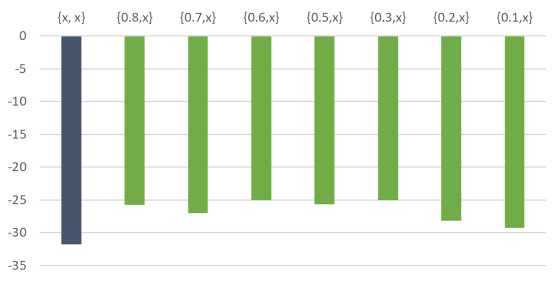
(f) HPWL relative to weights for CZ = 50%

Figure A.12: Impact of PWE_{fast} weight funtion on design metrics

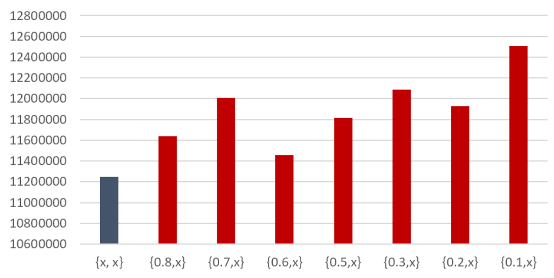
A.5 b19



(a) TNS relative to weights

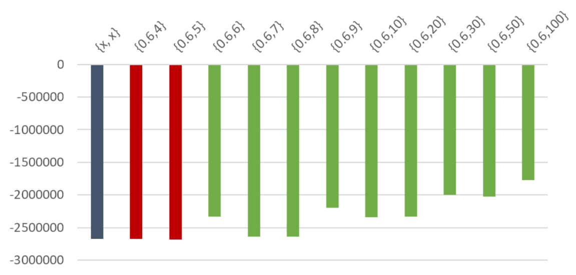


(b) WNS relative to weights

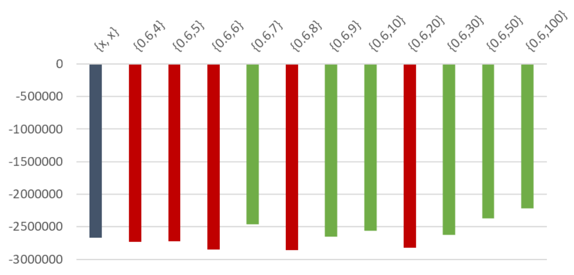


(c) HPWL relative to weights

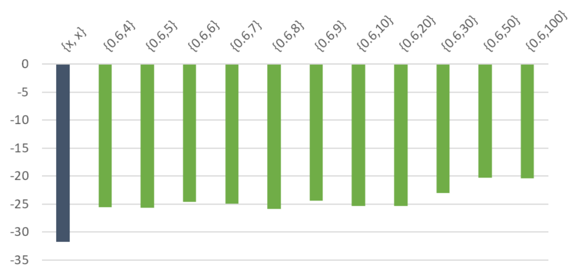
Figure A.13: Impact of *Linear* function on design metrics



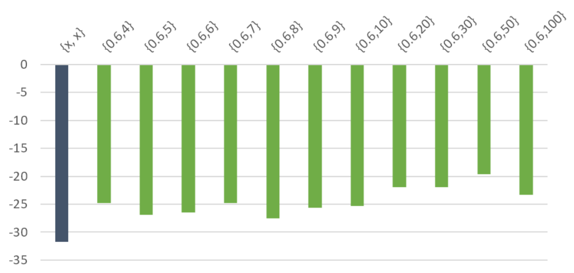
(a) TNS relative to weights for *PWL*



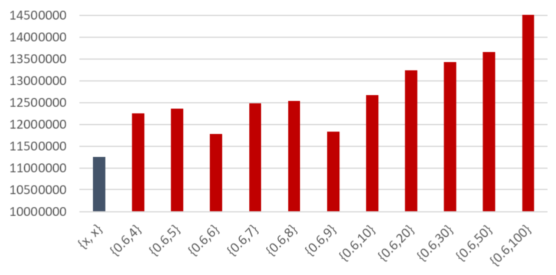
(b) TNS relative to weights for PWE_{slow}



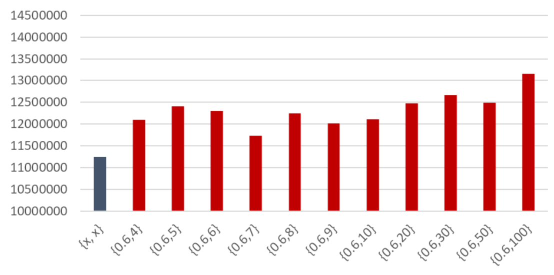
(c) WNS relative to weights for *PWL*



(d) WNS relative to weights for PWE_{slow}

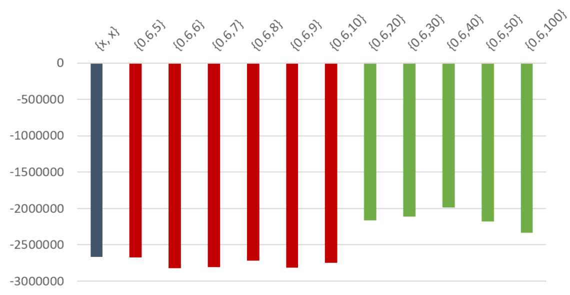


(e) HPWL relative to weights for *PWL*

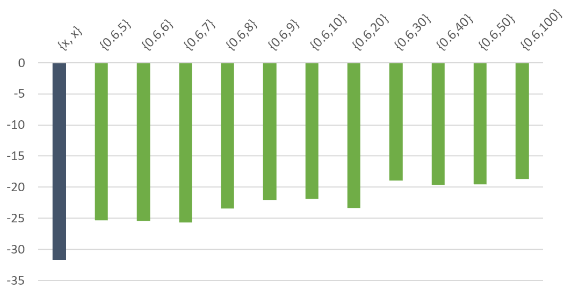


(f) HPWL relative to weights for PWE_{slow}

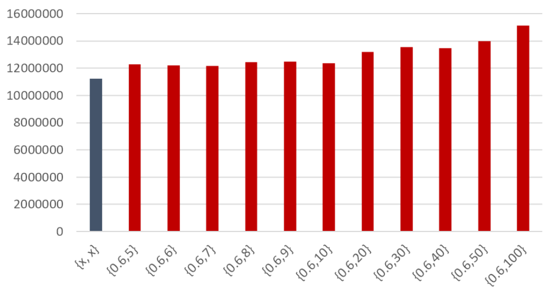
Figure A.14: Impact of *PWL* & PWE_{slow} weight functions on design metrics



(a) TNS relative to weights for CZ = 50%



(b) WNS relative to weights for CZ = 50%



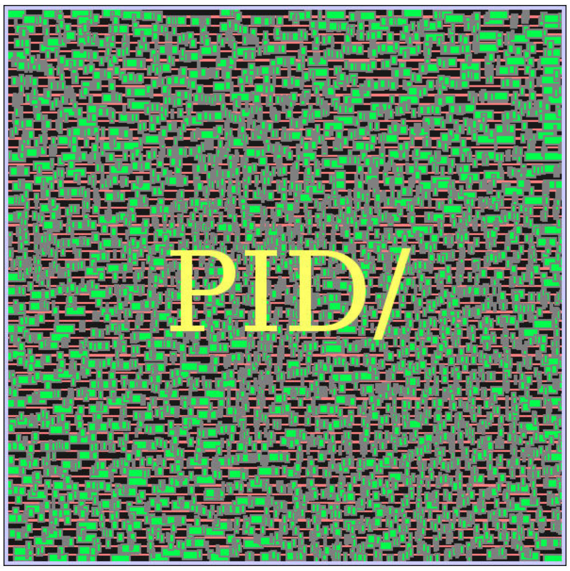
(c) HPWL relative to weights for CZ = 50%

Figure A.15: Impact of PWE_{fast} weight funtion on design metrics

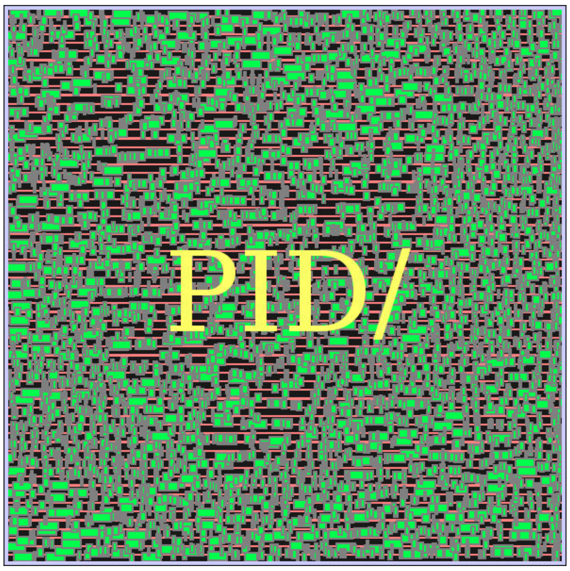
Appendix B

Design Layouts

B.1 PID

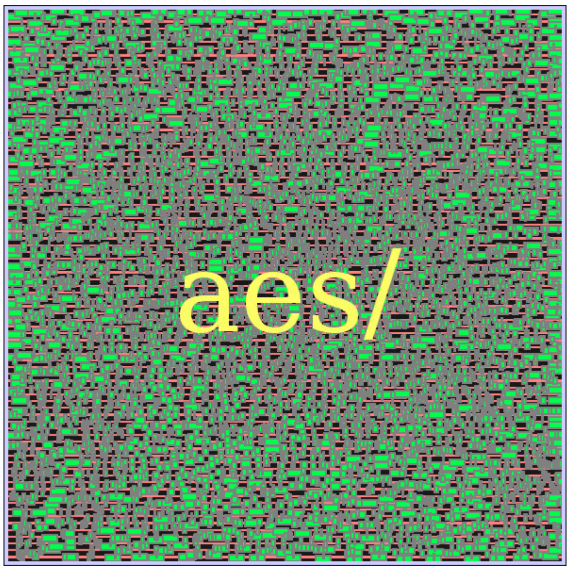


(a) PID non Timing-Driven Placement

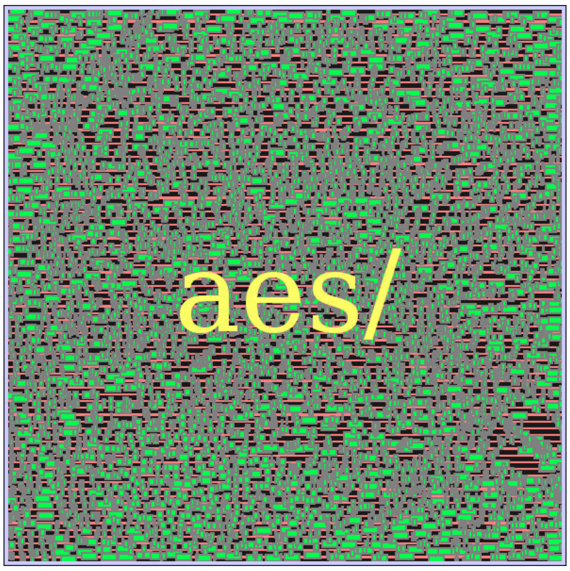


(b) PID Timing-Driven Placement

B.2 AES

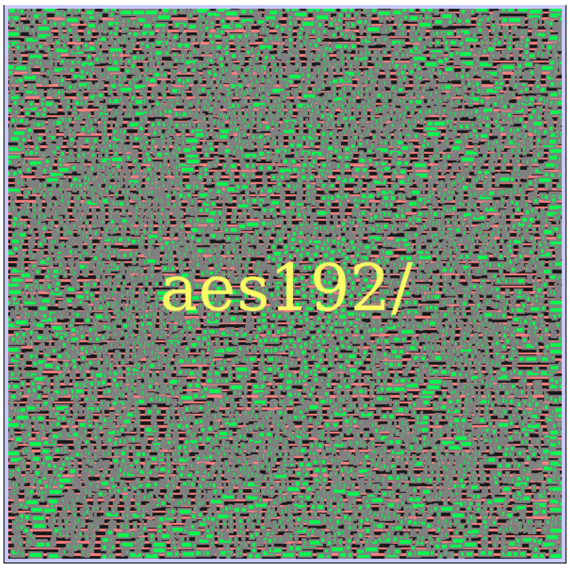


(a) AES non Timing-Driven Placement

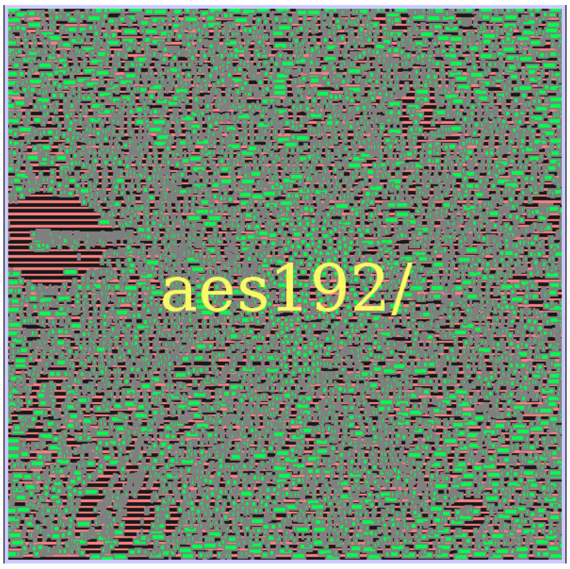


(b) AES Timing-Driven Placement

B.3 AES192

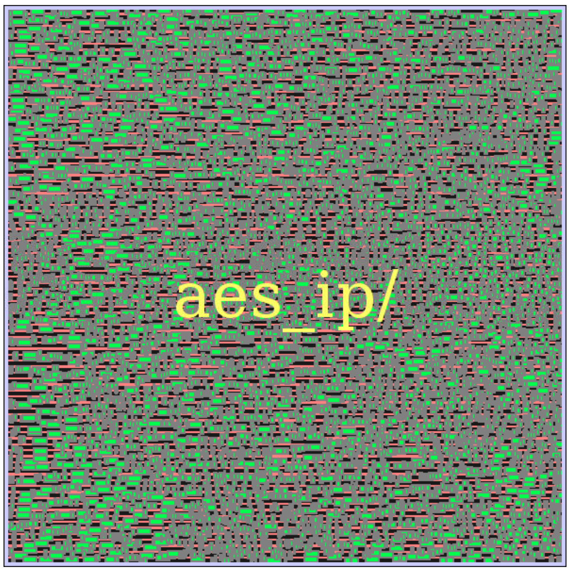


(a) AES192 non Timing-Driven Placement

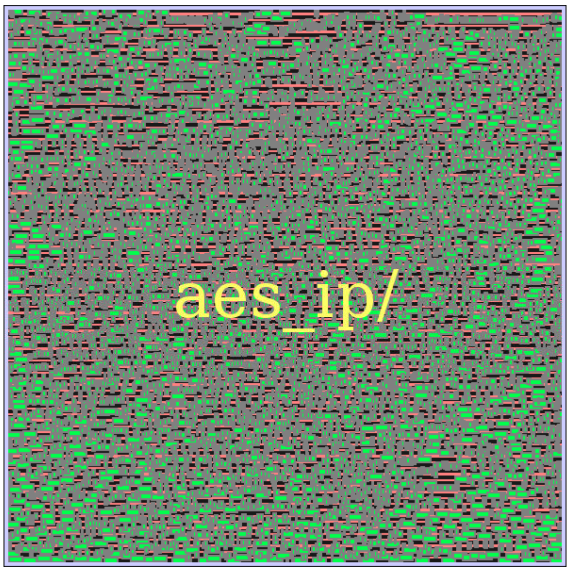


(b) AES192 Timing-Driven Placement

B.4 apbAES128

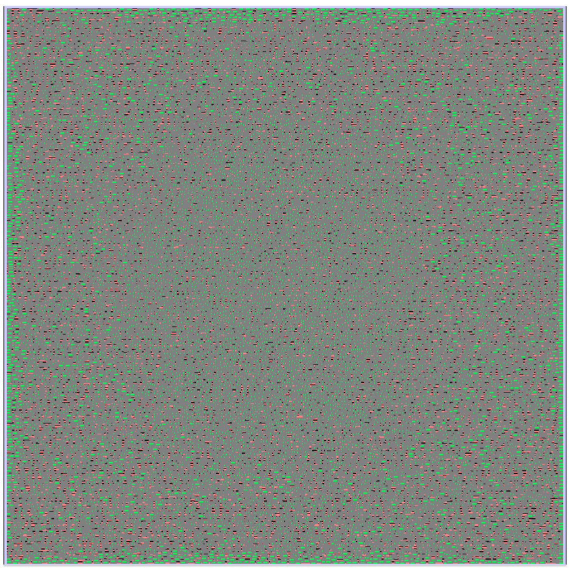


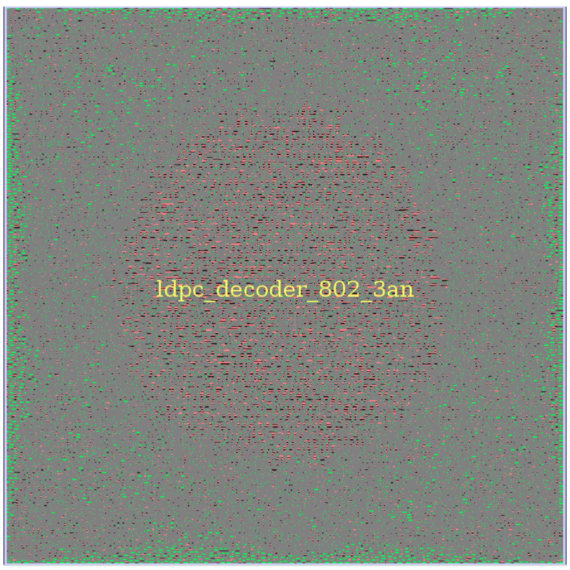
(a) apbAES128 non Timing-Driven Placement



(b) apbAES128 Timing-Driven Placement

B.5 LDPC


(a) LDPC non Timing-Driven Placement



(b) LDPC Timing-Driven Placement

Bibliography

[1] 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.

[2] P. Spindler and F. Johannes, *Kraftwerk: A Fast and Robust Quadratic Placer Using an Exact Linear Net Model*, 01 2007, pp. 59–93.

[3] K. M. Hall, “An r-dimensional quadratic placement algorithm,” *Management Science*, vol. 17, no. 3, pp. 219–229, 1970. [Online]. Available: http://www.jstor.org/stable/2629091

[4] N. Viswanathan and C.-N. Chu, “Fastplace: efficient analytical placement using cell shifting, iterative local refinement,and a hybrid net model,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 5, pp. 722–733, 2005.

[5] J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[6] S. S. S. Charles J. Alpert, Dinesh P. Mehta, *Handbook of Algorithms for Physical Design Automation.* CRC Press, Taylor & Francis Group, 2009.

[7] T. Kong, “A novel net weighting algorithm for timing-driven placement,” in *IEEE/ACM International Conference on Computer Aided Design, 2002. ICCAD 2002.*, 2002, pp. 172–176.

[8] H. Chang, E. Shragowitz, J. Liu, H. Youssef, B. Lu, and S. Sutanthavibul, “Net criticality revisited: An effective method to improve timing in physical design,” 08 2003.

[9] H. Youssef, R.-B. Lin, and E. Shragowitz, “Bounds on net delays for vlsi circuits,” *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 11, pp. 815–824, 1992.

[10] T.-Y. Wang, J.-L. Tsai, and C. C.-P. Chen, “Sensitivity guided net weighting for placement driven synthesis,” in *Proceedings of the 2004 international symposium on Physical design*, 2004, pp. 124–131.

[11] B. Riess and G. Ettelt, “Speed: fast and efficient timing driven placement,” in *Proceedings of ISCAS’95 - International Symposium on Circuits and Systems*, vol. 1, 1995, pp. 377–380 vol.1.

[12] R. Nair, C. Berman, P. Hauge, and E. Yoffa, “Generation of performance constraints for layout,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 8, pp. 860–874, 1989.

[13] K. Rajagopal, T. Shaked, Y. Parasuram, T. Cao, A. Chowdhary, and B. Halpin, “Timing driven force directed placement with physical net constraints,” in *Proceedings of the 2003 International Symposium on Physical Design*, ser. ISPD ’03. New York, NY, USA: Association for Computing Machinery, 2003, p. 60–66. [Online]. Available: https://doi.org/10.1145/640000.640016

[14] Q. B. Wang, J. Lillis, and S. Sanyal, “An lp-based methodology for improved timing-driven placement,” in *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.*, vol. 2, 2005, pp. 1139–1143 Vol. 2.

[15] P. Spindler, U. Schlichtmann, and F. Johannes, “Kraftwerk2—a fast force-directed quadratic placement approach using an accurate net model,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27(8), pp. 1398 - 1411, 09 2008.