UNIVERSITAT POLITÈCNICA DE CATALUNYA
DEPARTAMENT DE LLENGUATGES I SISTEMES INFORMÀTICS
MÀSTER EN COMPUTACIÓ

TESI DE MÀSTER

CLUSTERING FOR THE OPTIMISATION OF ASYNCHRONOUS CONTROLLERS

ESTUDIANT: Jonàs Casanova
DIRECTOR: Jordi Cortadella

DATA: 25 de Juny de 2008

Preface

This master's thesis is on the field of computer science applied on electronics. The miniaturisation of integrated circuits is bringing new problems in terms of power consumption, speed, and variability tolerance. The current synchronous designs are struggling to cope with these problems, and in consequence new optimisations or paradigms are being studied.

The study of this thesis are the optimisations like clock skew for synchronous circuits and asynchronous circuits as an alternative paradigm. The performance analysis of both cases are equivalent and algorithms on graph theory for cycles have been implemented to calculate the optimum speed.

Asynchronous controllers are essential for a good asynchronous design. To create a connectivity structure of controllers it is necessary to group the memory elements (registers) of the circuit into clusters. Clustering registers affects power consumption, performance, area, and variability tolerance. To produce a good clustering is a hard job because of the high number of registers and for the trade-offs of optimising all these characteristics.

An initial problem in clustering of controllers is to decide how many controllers we want. A design with one cluster give us the same problems of a synchronous design, high power consumption and too much sensible on variability of temperature, voltage, manufacturing errors, etc. On the other hand, having as many controllers as registers will produce too much overhead in area for all the new logic and wires that needs to be added.

It is important to have clusters as less connected as possible to design simple controllers and to minimise the impact on area. We know from benchmarks and industrial designs that the register graph is highly connected, and the controllers graph is almost complete. A variation of Min-Cut can give us a solution to optimise this property.

The clustering will have an impact on performance. Grouping registers implies a lost of freedom, and optimisations like clock skew or the asynchronous circuit will be affected by this lost as a handicap to reach the maximum speed.

From the placement point of view we need to have clusters where their registers are close to minimise the clock tree. The ideal solution is a partition of the space. The worst solution is to have the registers spared around.

The contribution of this thesis are two clustering algorithms; A local search solution to minimise the number of connections, and a k-means implementation that combines the minimisation of the clock trees and the maximisation of performance, by using parameters to balance it.

These algorithms have been implemented in the *Elastix* EDA tool and executed on ISCAS benchmarks and SUN Microsystems OpenSparc processor.

Acknowledgements

Several people has helped me making this thesis possible. I would like to thank Prof. Jordi Cortadella to give me the opportunity to work on this project, and to guide me from earlier years of my degree to here; Ferran Martorell to introduce me into the electronic design world and make me feel safe on it; and my family and friends to support me and made my life easy these months.

This research was supported by a grant from *Elastix Corporation*, thanks to an agreement with Universitat Politècnica de Catalunya. I would like to specially thank *Elastix Corporation* for the resources that had put on me to make this project.

Jonàs Casanova Bachs

Contents

Chapter 1

Introduction

This chapter starts with some history and current status of microelectronics and introduces generic concepts like synchronous and asynchronous circuits. It also introduce concepts about circuit performance that will be required to describe the statement of the problem.

1.1 Microelectronics

Integrated circuits experimented a revolution since VLSI (Very-large-scale integration) came in the 70's. This technology allowed to design complex circuits on a single chip with hundreds of millions of transistors. This revolution is compared to Industrial revolutions (steam machines first and electricity later) in a way that it speeds up the things we can do and introduce new possibilities such as communications, computations, sensors, information systems, etc This technology came with Computer Aided Design (CAD) tools that made all this possible.

Electronic Design Automation (EDA) tools aid engineers in all abstraction levels of the design. From high level specification to the physical implementation. From the computer science point of view all these steps are related to computational problems:

Microarchitecture gives a description of all differnet components of the circuit. For example, microprocessors have typically the Von Neumann architecture with: memory, control unit and arithmetic logic unit.

Logic synthesis transforms combinational logic expressions into implementation logic gates. This is specified by Register Transfer Level (RTL) description for synchronous circuits. It is the typical input for an EDA tool.

Verification tools are proof systems in several areas. In logic synthesis it is used to ensure that the implementation is consistent with the specification, and it can be used to verify the specification itself.

Static Timing Analysis (STA) estimates the delay of the circuit and detects possible timing errors. This information determine the speed of the circuit and it can be used for optimisations in the logic synthesis process.

Placement is the step where the circuit is described as a collection of components from a physical library, and it needs to set the postion of all these components on the die. This is an optimisation problem that tries to place close elements hat are logically connected.

Routing is one of the final steps and it comes after placement. Placement produce the position of each component on a die, but this components are connected with metal wires in several layers. The total wire length needs to be minimised with a fix number of layers depending on the technology used.

This thesis falls into the STA phase of the process, but having in mind the synthesis of physical layout (placement and routing) process that comes after. The next sections describe important concepts for the STA phase and presents the statement of the problem.

1.2 Synchronous circuits

Synchronous circuits are those in which the registers that separate combinational logic are controlled by a clock signal. Typically, these combinational blocks are described using a high-level language for hardware specification like Verilog or VHDL. This high-level description (RTL) will be transformed by an EDA tool to a low-level implementation using logic gates from a desired technology. At this point, it is possible to analyse the maximum and minimum delay that these logic will take to stabilise the result. Combinational blocks have a different delay to stabilise the result depending on the inputs. For example, a simple adder can take longer depending if it needs to propagate the carry or not. All registers store the computed value simultaneously guided by a clock signal. For the circuit to work correctly it requires that the clock signal arrives after the computation has stabilised the result. Figure 1.1 shows an example of a synchronous circuit.

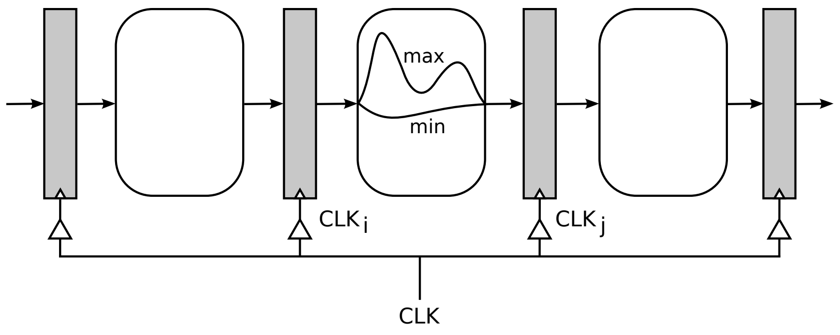


Figure 1.1: Synchronous circuit with a global clock. Showing min and max delays of the combinational blocks.

1.3 Asynchronous circuits

Synchronous systems had been chosen as a main design because of its simplicity to understand, verify and implement, compared to asynchronous systems. The fact of having a clock signal that synchronise all registers allows to work with discrete time units, and most of the EDA tools use this assumption to simplify the STA. The current high level of miniaturisation, 65nm and beyond, is making the industry to re-think old decisions and find solutions for new problems to keep improving performance. Things like power usage, speed, delay variability, Electromagnetic interference (EMI),... are becoming crucial. Asynchronous systems [1] are less attractive from the engineering point of view, but has solutions to all these new problems that high miniaturisation is bringing. Asynchronous designs can consume less power because of its natural way of *to work just if there is something to do*. The lack of a clock signal impacts on EMI positively. The system is more tolerant to temperature or voltage variation, thanks to the distributed control system (compared to the synchronous global clock signal)

In asynchronous systems combinational blocks are controlled by protocols. These protocols uses *requests* signals to initiate a computation and the corresponding *acknowledgements* to signal completion of that computation. The protocol knows the delay of the action from the timing analysis, and it can implement the signals to match the same delay. These signal delays can be implemented by counting discrete periods provided by a clock (used in the first dataflow computer, DDM1 [2]). This delay can also be implemented by a delay element as an inverter chain, which behave under the same conditions under variability. All conditions that may affect the computational delay of the combinational block it also affects the delay element. These are variations such as temperature, voltage, manufacturing precision, etc.

This protocol is implemented by controllers with some combinational logic and some delay elements. These controllers adds an extra complexity into asynchronous circuits compare to synchronous ones, where everything is controlled by a clock signal.

Figure 1.2 shows an example of an asynchronous circuit, with no global clock signal.

1.4 De-synchronisation

Going from synchronous into asynchronous implies changes on the design process. There are already EDA tools to design asynchronous circuits but the acceptance in the industry is low because it changes completely the EDA flow and it makes harder the specification phase. Recently, a new concept has appeared called *De-synchronisation* [3] that mixes both paradigms. The design process can be done on the synchronous world, using current synchronous EDA flow, and transform it into an asynchronous design automatically.

On that process the clock signal is removed and all registers will be assigned to handshake controllers. These controllers “know” the delay of the logic and send acknowledge and request signals to their neighbour controllers.

The verification process is not affected because *De-synchronisation* does not affect the combinational logic, and the same techniques used for synchronous circuits can be applied for de-synchronised ones.

Figure 1.2 shows the de-synchronisation of the circuit of Figure 1.1. In this transformation, the controller C_i sends the request for computation complete req signal to C_j. This signal is delayed to match the computation time. Once C_j has received the req signal, it can send the acknowledge signal back while it is saving the result to be used for the next register. This is just a simplification of the transformation, a real de-synchronisation involves a more complex protocols for the controllers.

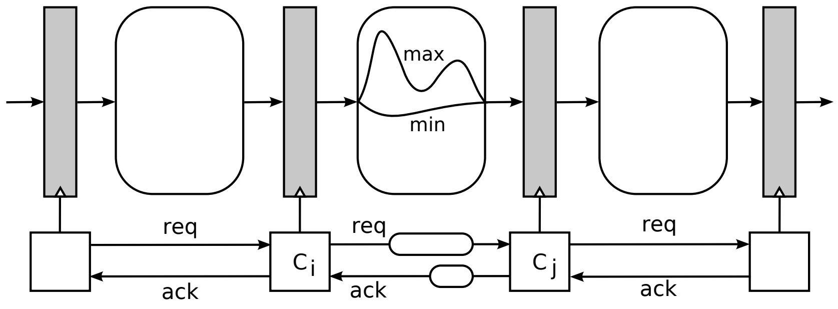


Figure 1.2: De-synchronisation of the circuit from Figure 1.1.

1.5 Performance

There are many ways to measure the performance of a circuit. From the architectural point of view there are metrics to measure performance like *IPC: Instructions Per Cycle* or *FLOPS: FLoating point Operations Per Second*. These metrics are used to compare different architectures. The study of this thesis is not based on the architectural point of view, it is based on the static timing analysis (STA).

Synchronous circuits uses memory components called registers like flip-flops and latches to synchronise all combinational paths. These paths determine the delay of the circuit, and the performance of a circuit is given by the minimum period of this synchronisation signal.

Asynchronous circuits cannot be studied as a collection of paths because there is no synchronisation signal. STA needs to evaluate all cycles to determine the maximum mean cycle.

Retiming

Retiming [4] is a period optimisation technique for synchronous circuits. It moves registers across the logic to minimise the maximum delay between registers. It can reduce the minimal period while keeping the same sequential behaviour.

Figure 1.3 shows a circuit where the register X is retimed to reduce the delay between A and X. The resulting circuit has a maximum delay between registers of 2 time units, instead of the original 3 time units. The retiming process can introduce new registers as in the example like X_1 and X_2. These new registers increase the area of the circuit and the power consumption. However, on this example it is possible to apply retiming on the register Y and get the same number of registers back and keeping the optimum delay of 2 time units (Y and X_1 will be storing the same information). Notice how the output of register R will not change.

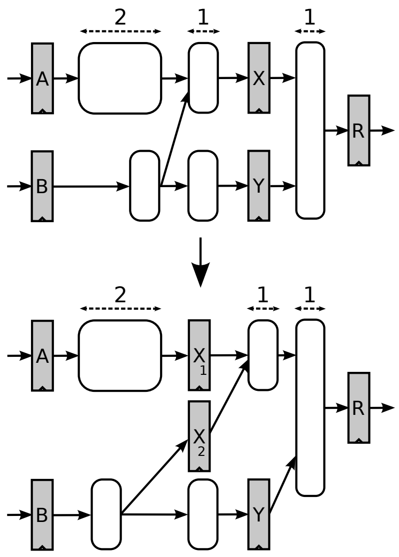


Figure 1.3: Retiming example for register X.

Retiming is not used on this thesis, clock skew optimisation is used instead because its implicit application on *De-synchronisation*.

Clock Skew Optimisation

Clock Skew is defined as the delay that occurs from the source of the clock signal to the final register. The clock signal is propagated via wires and gates forming a tree, mesh, or other structures, and it implies some delay. Traditionally this has been seen as a problem. The delay of distributing the clock had to be minimised and the period had to be augmented to cope with this margin of time between registers.

Clock Skew Optimisation is a technique to improve the period for synchronous circuits introduced by Fishburn [5] from the basic idea that registers can give or borrow time to neighbours to minimise the clock period. In this new design, the clock signal of each register can be delayed to match the timing requirements of the logic. This delay can be inserted on the clock line before the register like in Figure 1.1. This optimisation extends the timing analysis including hold and setup constraints for all paths.

Having a period P, any connected registers i and j with skews X_i and X_j, and the minimum and maximum delay between i and j as MIN_{ij} and MAX_{ij} :

Setup Constraints make sure that any two connected registers have enough time to stabilise the result before j stores the result.

$$Setup\,Constraint : MAX_{ij} \leq P + X_j - X_i = Setup\,time$$

Hold constraints avoid that the next cycle of i will overwrite the data of the current one which j still has to store.

$$Hold\,Constraint : MIN_{ij} \geq X_j - X_i = Hold\,time$$

Figure 1.4 shows a timing diagram were *Setup Constraint* and *Hold Constraint* can be graphically interpreted. A *Setup Constraint* shows how the $max\ delay$ can be longer than the period thanks the skew applied on register j. A *Hold Constraint* shows how the $min\ delay$ cannot arrive before the second rise edge of j, otherwise it will overwrite the value of the fist wave ($max\ delay$). Notice how the logic is processing 2 waves at the same time.

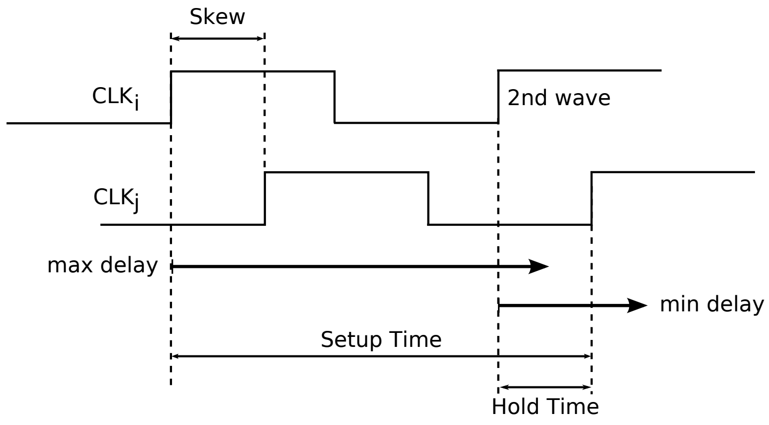


Figure 1.4: Clock diagram of two registers i and j. Showing $max\ delay < Setup\ time$ and $min\ delay > Hold\ time$.

Joy D.A [6] introduced *Logic Signal Separation Constraints*, these constraints adds relations between all predecessor nodes for any node. One predecessor can overwrite data from another predecessor depending on the $min\ delay$ that share. To avoid this problem Joy adds new constraints similar to *Hold Constraints* between these nodes.

The problem of minimising the clock period of a circuit by optimising clock skews can be solved using Linear Programming [5, 6, 7]. Having a collection of *Setup Constraints* and *Hold Constraints* where the Period has to be minimised. It also can be solved using a binary search [8, 9, 10, 11]. This thesis uses the binary search approach in Chapter 2 Section 2.7.

Clock Skew and Retiming

Retiming and Clock Skew are both continuous and discrete optimisations with the same effect. This fact was already observed in [5]. However, these two techniques can be combined to obtain better results.

Retiming can be used as a post process of Clock Skew Optimisation to reduce the skew difference between registers [12]. The idea is to find the optimum period and assign skews with an small difference between the minimal skew and the maximal skew (Using the same algorithm as in Chapter 2 Section 2.7). In a second phase, it minimise the skew difference by applying retiming. It is important to start from a solution with small differences between clock skews because less retimings will be needed to reduce the time difference.

Clock Skew and Retiming can be used in combination to improve the clock period [13]. This paper shows how the combination can do better than both techniques separately. The idea is that Retiming can help to fix hold constraints and the period of Clock Skew Optimisation can be improved. The problem is solved by a mixed-integer linear programming formulation and an heuristic. Chapter 2 describes the implications of using hold constraints or not, and Chapter 4 shows some results of these differences.

1.6 Statement of the Problem

This thesis is situated on the STA phase of the design flow. The motivation comes from the necessity to clusterise registers on the *De-synchronisation* process. The interest from the industry reflects the importance of the problem. This thesis uses graph theory to analyse circuit performance assuming clock skew optimisation or asynchronous circuits. The contributions of this thesis are algorithms for the clusterisation of registers for synchronous and asynchronous systems. All algorithms had been designed and integrated in the *De-synchronisation* flow of *Elastix Corporation* EDA tool.

There are several cluster properties that will be studied and optimised. The next paragraphs describes what is a cluster and the properties.

Clustering

The circuit is the starting point and it is represented by a *Register Graph* where each node represents a register and directed edges represent combinational blocks. R is the number of nodes of this graph.

A **Cluster** is a Set of registers. In an asynchronous circuit, it can represent the nodes under the same controller, and in a synchronous circuit it can represent nodes that have the same skew as in [14].

A *Cluster Graph* is constructed from the *Register Graph* in a way that nodes on the *Cluster Graph* represent partitions of the *Register Graph*. Two clusters are connected in the *Cluster Graph* if any of their registers are connected.

A Clustering can vary from 1 partition solution to R partitions (one for each register). A Clustering solution has several properties with a high design impact. Clustering based on *Connections*, *Performance*, and *Placement* are studied on this thesis and described on the next sections.

Connections

The number of connections is defined as the number of edges on the cluster graph. It is important to minimise this number to reduce the complexity of the controllers. Having many connections implies to increase the number of delay lines, logic, and metal wires to connect controllers. This may have consequences on area usage as well.

From the point of view of cluster connection, it does not matter if there is one or more registers connected between clusters, it only matters connections at cluster level. It is not the same minimising the number of connections on the *Cluster Graph* than minimising the number of registers connected to different clusters.

The problem of minimising the number of registers connecting differnet clusters can be seen as the *MIN-CUT* problem of the *Register Graph*. hMetis [15] is a multilevel partitioning algorithm that performs very well for this problem.

A local search solution is presented on the Chapter 3 Section 3.1 that minimise the number of cluster connections and it can be easily extended with other cost functions (Future work).

Performance

Performance analysis of a cluster solution can be seen as a synchronous circuit with clock skew optimisation where only few skews can be applied. Each skew represents a cluster. Having the Cluster Graph with maximum and minimum delay annotated on the edges, performance is determined as the minimum period can be calculated using the same algorithm that the one used for the Register Graph.

Finding the best performance for a given number of clusters is the same problem as Multi-domain clock skew scheduling [14] solved using a branch-and-bound search using a SAT solver. According to the authors this algorithm takes 20 hours for an industrial design of 250737 edges. This thesis expects to work with circuits of 400k edges and much more. A better heuristic is required to deal with bigger circuits.

A clustering solution that optimise performance based on k-means [16, 17, 18] clustering (combined with placement) is presented on the Chapter 3 Section 3.2.

Placement

Clustering in terms of placement groups registers using their position information on the die. The goal is to minimise the cost of connecting all registers of the same cluster. The problem of calculating the cost for a given cluster can be seen as calculating the Steiner tree [19] of these nodes. A Steiner tree connects all nodes and it can introduce extra connection points to minimise the total length. This is an NP-complete problem, but Spanning trees simplifies Steiner trees because no extra connection points can be added. Two traditional algorithms to calculate spanning trees are Prim [20] and Kruskal [21]. Both of them are linear and are a good approximation compared to the Steiner tree NP cost. Another typical metric is the half perimeter wire length (HPWL) calculated as the half perimeter of the bounding box that include ll registers of the cluster. HPWL is very fast to calculate but the error committed can be huge compared to trees methods.

A clustering solution that optimise placement based on k-means [16, 17, 18] (combined with performance) is presented on the Chapter 3 Section 3.2

Tradeoff

A good clustering algorithm needs to face the tradeoff between optimising one or the others properties. A perfect clustering for performance could be a total disaster in terms of placement and become an impossible design to implement due to the routing overhead. A perfect clustering for placement could probably become a difficult design to optimise with clock skew.

This thesis presents a clustering algorithm that combines performance and placement to generate an implementable design with good properties.

1.7 Description of the Chapters

Chapter 2 reviews a solution to calculate the period of a circuit and describes how it has been implemented on *Elastix Corporation* EDA tool.

Chapter 3 presents the main contribution of this thesis: the algorithm for clustering that optimises the number of connections, and the algorithm for clustering optimising placement and performance.

Chapter 4 presents results of several experiments using ISCAS benchmarks and SUN's OpenSparc processor.

Chapter 5 presents the conclusions of this thesis and future work.

Chapter 2

Optimal period with clock skew

The next sections will explain how to calculate the period of a sequential circuit and how to improve it using clock skew optimisation. The chapter uses the same circuit example (Figure 2.1) to introduce concepts and apply optimisations.

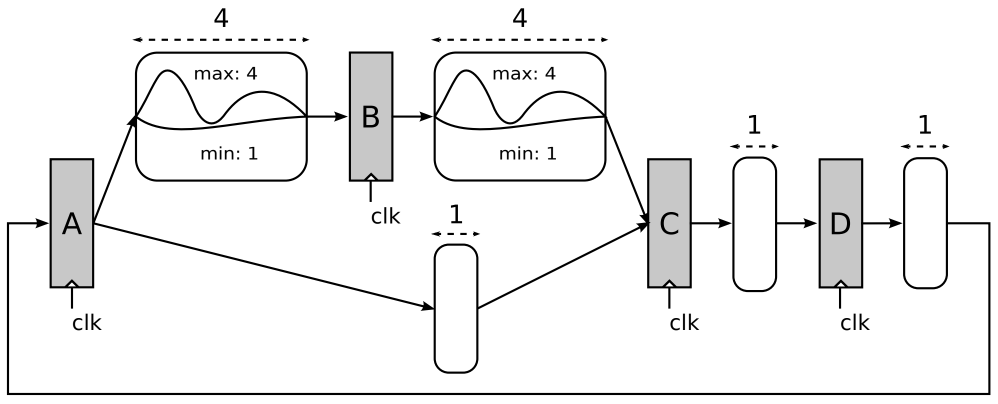


Figure 2.1: Basic circuit example with minimum and maximum delays on combinatorial paths.

2.1 Delay Graph

The data structure used to describe a circuit is a graph. A Delay Graph is a representation of the combinational blocks of the circuit with timing information annotated on the edges.

- Nodes in the Delay Graph are Inputs, Outpus, or Registers.

- Directed edges in the Delay Graph are combinational paths between nodes through logic gates. Each edge has the minimum and maximum delay of the combinatorial logic that represents.

This data structure can be used for the Static Timing Analysis (STA) to calculate the optimum period of the system. For example, the delay graph of the example circuit of Figure 2.1 can be seen in Figure 2.2. In a simple synchronous circuit all registers are synchronised by a clock signal. The period of that clock will be determined by the maximum delay. In this example, the maximum delay is 4 and if no optimisations are applied, the corresponding synchronous circuit will have a Period of 4.

Next sections will improve this period by applying some optimisations on the clock signal.

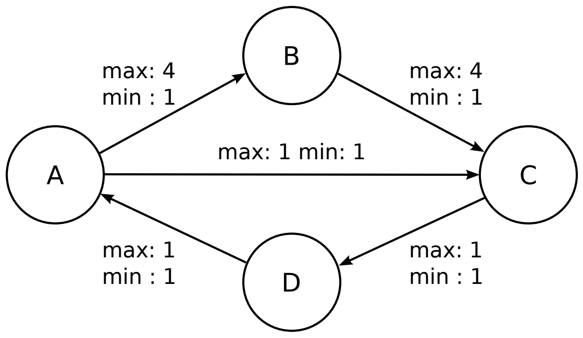


Figure 2.2: Delay graph example with min/max delay

2.2 Clock Skew Optimisation

Clock Skew Optimisation [5] (Introduced on Chapter 1 Section 1.5) is a technique that can improve the period of a circuit by applying a delay on the clock signal of each register. It basically defines a set of Setup and Hold Constraints that need to be satisfied.

Having a period P, any connected registers i and j with skews X_i and X_j, and the minimum and maximum delay between i and j as MIN_{ij} and MAX_{ij} :

Setup Constraints make sure that any two connected registers have enough time to stabilise the result before j stores the result.

$$Setup\,Constraint : MAX_{ij} \leq P + X_j - X_i = Setup\,time$$

Hold constraints avoid that the next cycle of i will overwrite the data of the current one which j still has to store.

$$Hold\,Constraint : MIN_{ij} \geq X_j - X_i = Hold\,time$$

One possible simplification of the clock skew optimisation is to eliminate the *Hold Constraints*. Assuming that it is possible to add delays on the minimum paths to fix all hold violations without increasing the maximum delay path. This new problem with just *Setup Constraints* is equivalent to calculate the *Maximum Mean Cycle (MMC)* of the graph using the max delays.

2.3 Maximum Mean Cycle

Having a directed Graph $G = (V, E)$ with max delays annotated on the edges. Let C be a cycle in G, where $|C|$ is the number of edges and $w(C)$ is the weight of the cycle (sum of maximum delays of the cycle edges):

$$Maximum\ Mean\ Cycle = \max_{C \in G} \left\{ \frac{w(C)}{|C|} \right\}$$

On Figure 2.2, 2 cycles exist:

$$\begin{aligned} Mean\ Cycle(A \rightarrow B \rightarrow C \rightarrow D \rightarrow A) &= \frac{4+4+1+1}{4} = 2.5 \\ Mean\ Cycle(A \rightarrow C \rightarrow D \rightarrow A) &= \frac{1+1+1}{3} = 1 \\ Maximum\ Mean\ Cycle &= \max(2.5, 1) = 2.5 \end{aligned}$$

The maximum period without using clock skew of the circuit in Figure 2.1 is 4 (max delay). Figure 2.3 shows the same circuit but applying clock skew to achieve a period of 2.5 (ignoring hold constraints). The register B has been delayed 1.5 time units to be able to compute the maximum time between A and B. The same situation happens on register C, it need an extra 1.5 units to have enough setup time to compute the maximum delay. The registers B and C can borrow 1.5 time units from $C \rightarrow D$ and $D \rightarrow A$, and give it to $A \rightarrow B$ and $B \rightarrow C$ because they need it to compute the max delay ($2.5 + 1.5 = 4$).

Figure 2.4 shows a timeline for a period of 2.5 with the following observations:

- It shows 2 complete waves through the cycle. Each cycle takes 4 periods, and thanks to clock skew all 10 time units can be accommodated. All combinatorial blocks finish their computation before the rising edge of the ending register.
- The min delay of $A \rightarrow B$ and $B \rightarrow C$ needs to be increased to not override the previous wave.
- The min and max delay of $A \rightarrow C$ needs to be increased a significant amount of time. This example shows the extreme case of having a clock skew bigger than the period, and how a small edge can be affected.
- This example shows how it is possible to reduce the period by adding delays.

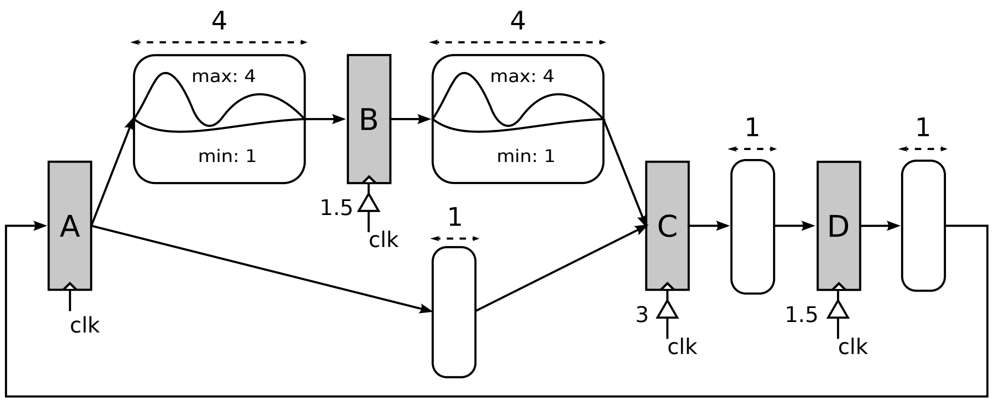


Figure 2.3: Basic circuit with clock skews ignoring hold constraints. Period of 2.5.

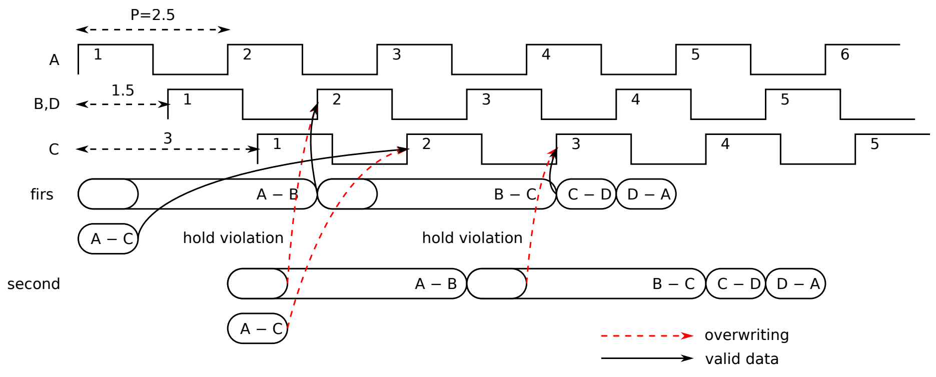


Figure 2.4: Timeline for a period of 2.5, showing hold violations that can be fixed by incrementing the delays.

2.4 Maximum Mean Cycle algorithms

There are several algorithms to compute the MMC. The most popular algorithms are Krap's algorithm [22], Lawler's algorithm [23], Burns' algorithm [24], and an improved version of Howard's algorithm [25]. A report [26] compares all of them in which Howard's algorithm appears to be the fastest one.

All of them are based on a parametric shortest path problem. Burns' algorithm uses linear programming but with a similar idea.

2.4.1 Parametric shortest path

The parametric shortest path problem is a generalisation of the single-source shortest path problem in which some of the edges have a parameter subtracted from them. The problem is to generate a shortest path tree with the maximum value of that parameter without generating any negative cycle. This problem is related to the maximum mean cycle problem because the value of this parameter is the MMC if the parameter is subtracted from all edges. It represents the maximum value it can be subtracted from all cycles until a cycle has a weight equals to 0.

An improved version of Krap's algorithm by Young,Tarjan,Orlin [27] shows how the parametric shortest path problem can be used to compute the maximum mean cycle as well as the minimum balance problem (used in Section 2.9).

This thesis uses Howard's algorithm to compute the MMC of the max delays of the circuit.

2.4.2 Howard's algorithm

All shortest path algorithms require a graph with a $weight$ annotated on the edges, and it uses a distance property $dist$ for nodes to set the shortest distance from the reference node.

Howard's algorithm uses the idea of the parametric shortest path problem but on the opposite way of the other algorithms. It starts with a parameter $\lambda = \infty$ and decreases the λ value until the graph is valid and no negative cycle exist. The algorithm maintains a *policy graph* which is a subgraph where each node has a out degree of one. This graph represents a current "shortest path tree". At each iteration, a shortest path is found using a breadth-first search (BFS) algorithm. If a valid path is found, λ is valid and it can be returned, or λ needs to be updated as well as all edges and the *policy graph*.

2.5 Constraint Graph

The problem of assigning valid skews to registers for a given period with *Setup* and *Hold Constraints* (and determines if the period is valid) can be solved by linear programming [5, 6, 7]. In fact, it belongs to the subclass *difference constraints* problem

Algorithm 1 Howard's Minimum Mean Cycle algorithm

```
Require: A strongly connected graph $G(V, E)$, $weight(e)$
Ensure: The minimum cycle mean $\lambda^*$ of $G$
 1: for all $u \in V$ do
 2:     $d(u) \leftarrow +\infty$
 3: end for
 4: for all $(u \rightarrow v) \in E$ do
 5:     if $weight(u \rightarrow v) < dist(u)$ then
 6:         $dist(u) \leftarrow weight(u \rightarrow v)$; $\pi(u) \leftarrow v$ {$\pi$ is the policy}
 7:     end if
 8: end for
 9: while $true$ do
10:     $E_\pi \leftarrow (u \rightarrow \pi(u)) \in E$ { Find the set of $E_\pi$ of policy arcs}
11:     Examine every cycle in $G_\pi = (V, E_\pi)$
12:     Let $C$ be the cycle with the smallest mean in $G_\pi$
13:     $\lambda \leftarrow weight(C)/ \mid C \mid$
14:     Select an arbitrary node $s \in C$
15:     { Compute the node distances using the revers BFS}
16:     if there is a path from $v$ to $s$ in $G_\pi$ then
17:         $dist(v) \leftarrow dist(\pi(v)) + weight(v \rightarrow \pi(v)) - \lambda$
18:     end if
19:     {Improve the node distances}
20:     $improved \leftarrow false$
21:     for all $(u \rightarrow v) \in E$ do
22:         $\delta(u) \leftarrow dist(u) - (dist(v) + weight(u \rightarrow v) - \lambda)$
23:         if $\delta(u) > 0$ then
24:             if $\delta(u) > \varepsilon$ then
25:                 $improved \leftarrow true$
26:             end if
27:             $\delta(u) \leftarrow dist(v) + weight(u \rightarrow v) - \lambda)$
28:             $\pi(u) \leftarrow v$
29:         end if
30:     end for
31:     if NOT $improved$ then
32:         return $\lambda$
33:     end if
34: end while
```

and it can be easily solved using a difference constraints graph. For example, this graph representation is used in [11] to minimise clock skews.

Difference Constraints Graph

Having a difference constraints problem where y and x are variables, and K are constants :

$$\begin{aligned} y_1 - x_1 &\leq K_1 \\ y_2 - x_2 &\leq K_2 \\ &\dots \end{aligned}$$

This inequality system can be transformed into a graph such that each node represents a variable x_1, y_1, x_2, y_2,... and for each related variables a directed edge is created with the corresponding weight as: $x_1 \xrightarrow{K_1} y_1$. If the resulting graph is not strongly connected and extra node v_0 needs to be added that connects to all nodes with weight 0.

An interesting property of Constraint Graphs is that the system is consistent if no negative cycle exists. It is possible to assign values to nodes that satisfy all constraints. This problem can be solved using a shortest path algorithm like Bellman-Ford[28, 29, 30]. This algorithm assigns values to the nodes that satisfy the system (if it is consistent) with complexity $O(N \times E)$.

Circuit Constraints Graph

It is possible to use a Difference Constraints Graph to check if a given period is consistent with all setup and hold constraints. Re-arranging the inequalities:

$$\begin{aligned} Setup\,Constraint : MAX_{ij} &\leq P + X_j - X_i \\ X_i - X_j &\leq P - MAX_{ij} \\ Hold\,Constraint : MIN_{ij} &\geq X_j - X_i \\ X_j - X_i &\leq MIN_{ij} \end{aligned}$$

A circuit constraint graph is build using these inequalities. There is no need to add the extra v_0 node because the resulting graph is strongly connected. If the register graph has just one component, the constraint graph will be strongly connected because hold and setup constraints adds edges in both directions for any connected registers.

After running a shortest path algorithm like Bellman-Ford, all nodes will have a distance assigned. This distance can be interpreted as a valid skew for the period P.

This graph can be complemented with other timing constraints. *Logic Signal Separation Constraints* introduced by Joy D.A [6] are constraints between all predecessor nodes for any node. Joy proved by simulation than one predecessor can interfere with another one . These constraints can be included on the graph as extra edges.

Figure 2.5 shows a constraint graph with setup and hold constraints of the basic example with an invalid period of 2.5, Figure 2.6 for an invalid period of 3, and Figure 2.7 for a valid period of 3.5.

2.6 Improved Bellman-Ford algorithm

Bellman-Ford algorithm [28, 29, 30] can be used to solve a Circuit Constraints Graph. It determines if a given period is valid and assigns skews to registers. An improved version from Tarjan [31] with negative cycle detection has shown very good results [32] and it has been implemented on this thesis.

The classical Bellman-Ford (Algorithm 2) has no special mechanism to detect a negative cycle in advance. Having a Graph $G = (V, E)$, Bellman-Ford iterates $n \times e$ times, where n and e are the number of vertexes and edges. After all iterations it has to check if there is any negative cycle. The check is done by exploring all edges to see if there is any shortcut. As all possible paths up to size $n - 1$ have been explored at that point and it is not possible to have a short path with a length higher or equal than n. There is a negative cycle if such path exists.

Algorithm 2 BellmanFord

```
Require: Graph $G(V, E)$,$dist(v)$,$weight(e)$
Ensure: Updated $dist(v)$ such that $\forall_{(u \to v) \in E} dist(v) - dist(u) \leq weight(u \to v)$
 1: for $i = 1$ to $size(V) - 1$ do
 2:     for all $(u \to v) \in E$ do
 3:         if $dist(v) > dist(u) + weight(u \to v)$ then
 4:             $dist(v) \leftarrow dist(u) + weight(u \to v)$
 5:         end if
 6:     end for
 7: end for
 8: for all $(u \to v) \in E$ do
 9:     {Negative cycle check}
10:     if $dist(v) > dist(u) + weight(u \to v)$ then
11:         return $G$ contains a negative cycle
12:     end if
13: end for
```

The classical Bellman-Ford behaves badly when a negative cycle exist because it needs to compute all $n \times e$ operations. It checks for a negative cycle at the end. However, it is possible to do the check at the same time that it is searching for shortcuts. Bellman-Ford can maintain a shortest path tree by updating the predecessor on the tree each time that it finds a shortcut. A technique called *Walk to the root* uses this tree each time that a shortcut has been found. The algorithm needs to check if this new path is forming a cycle on the tree. This walk increases the complexity of the algorithm because the "walk" to the root costs $O(n)$ and increases the bellman-Ford complexity to $O(n^2e)$ because is done at each iteration.

The Improved Bellman-Ford (Algorithm 3) uses the negative cycle detection *subtree disassembly* introduced by Tarjan [31]. This improvement is based from the *Walk to the root* technique but it amortise the cost of traversing the tree over the work to building it. The idea is to "forget" the new shortpath and mark the new node as unreached (line 17). In conclusion: the Improved Bellman-Ford algorithm has a complexity of $O(ne)$ and detects for negative cycles much sooner.

Algorithm 3 ImprovedBellmanFord

```
Require: Graph $G(V,E)$,$dist(v)$,$weight(e)$
Ensure: Updated $dist(v)$ such that $\forall_{(u\rightarrow v)\in E} dist(v) - dist(u) \leq weight(u \rightarrow v)$
 1: $Q1 \leftarrow \emptyset, Q2 \leftarrow \emptyset$
 2: for all $e \in E$ do
 3:   if $dist(v) > dist(u) + weight(u \rightarrow v)$ then
 4:     append $u$ to $Q1$
 5:   end if
 6: end for
 7: while $Q1$ not empty do
 8:   $u \leftarrow pop(Q1)$
 9:   for all $v$ adjacent to $u$ in $G$ do
10:     if $dist(v) > dist(u) + weight(u \rightarrow v)$ then
11:       delete subtree rooted at $v$
12:       if $u$ was in the subtree deleted above then
13:         return negative cycle detected
14:       else
15:         $dist(v) \leftarrow dist(u) + weight(u \rightarrow v)$
16:         make $v$ a child of $u$ {constructing subtree}
17:         append $v$ to $Q2$
18:       end if
19:     end if
20:   end for
21:   if $Q2$ is empty then
22:     return completed: $dist$ satisfies constraints
23:   else
24:     $Q1 \leftarrow Q2, Q2 \leftarrow \emptyset$
25:   end if
26: end while
```

2.7 Binary search to find the optimal period

It is possible to find the optimal period with hold and setup constraints by Linear Programming as in [5, 6, 7], or by binary search [8, 9, 10, 11]. The binary search approach has been chosen in this thesis.

An upper bound of the period is the maximum max delay (the period of the circuit without clock skew). A lower bound is the *Maximum Mean Cycle* explained on the previous section were Howard's [25] algorithm responds quasi linear on average case.

The binary search builds a constraint graph for a target period and uses the Improved Bellman-Ford (Algorithm 3) to test if that period is valid. It is important to use the improved version because the binary search expects to test invalid periods, and an invalid period is generating a constraint graph with a negative cycle.

Next Figures will show some periods between the lower and the upper bounds of the example.

- Upper bound is $\max_{e \in E} MAX_e = 4$
- Lower bound is $MMC = 2.5$

Figure 2.5 shows how the lower bound, a period of 2.5, is not a valid period using *Hold Constraints*. There are several negative cycles, for example: $A \rightarrow C \rightarrow B \rightarrow A$ with weight -2. It is not possible to assign skews to nodes that satisfy all constrains. This period is also used on the timeline of Figure 2.4 where violations can be interpreted.

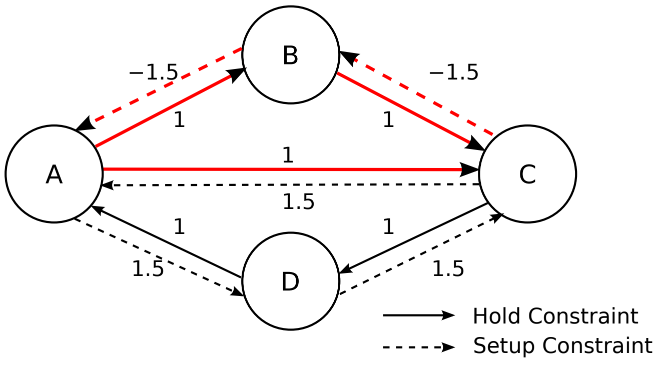


Figure 2.5: Constraint graph for period 2.5 . Negative cycle $A \rightarrow B \rightarrow A$ with weight -0.5 , $B \rightarrow C \rightarrow B$ with weight -0.5, and $A \rightarrow C \rightarrow B \rightarrow A$ with weight -2

Figure 2.6 shows how a period of 3, is not a valid period using *Hold Constraints*. There is a negative cycle $A \rightarrow C \rightarrow B \rightarrow A$ with weight -1. It is not possible to assign skews to nodes that satisfy all constrains, but it is possible to view how to fix this negative cycle. Increasing the min delay of one *Hold Constraint* $A \rightarrow C$ or decrease the delay of any of the *Setup Constraints* $C \rightarrow B$ or $B \rightarrow A$.

Figure 2.7 shows the constraint graph of the same example using a period of 3.5. In this case 3.5 is a valid period. This Figure shows the values calculated by Bellman-Ford algorithm starting from A.

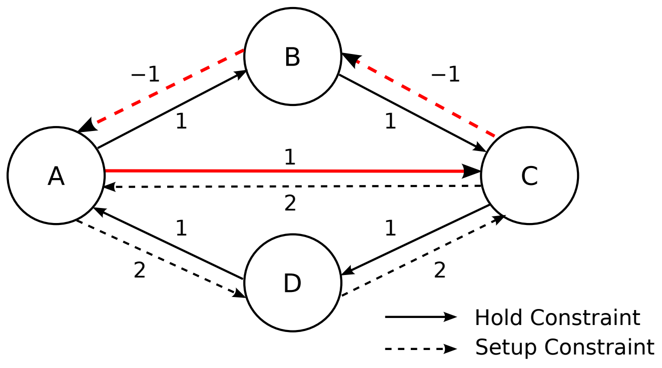


Figure 2.6: Constraint graph for period 3. Negative cycle $A \rightarrow C \rightarrow B \rightarrow A$ with weight -1

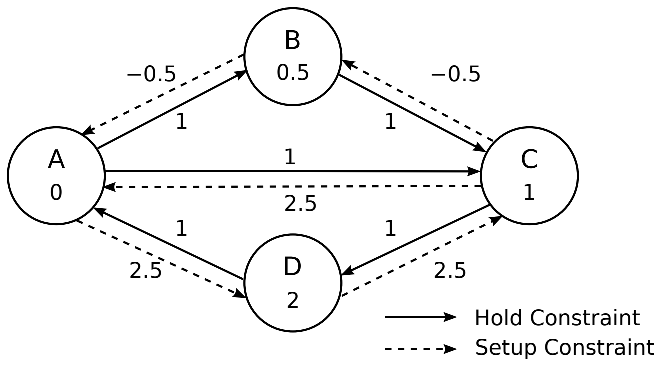


Figure 2.7: Constraint graph for period 3.5 with Bellman-Ford distances (skews).

2.8 Slack Graph

The slack graph is isomorphic to the constraint graph with different information on the edges. Once the skew has been assigned to all nodes, the slack is the maximum time the nodes can still differ from each other without breaking any constrain.

The slack of an edge between i and j is $S_{ij} = X_i + W - X_j$ and it is always positive if the assigned skews are valid.

Figure 2.8 shows the example with slack values for the 35 Period solution. Notice how the critical cycle $A \to C \to B \to A$ has 0 accumulated slack.

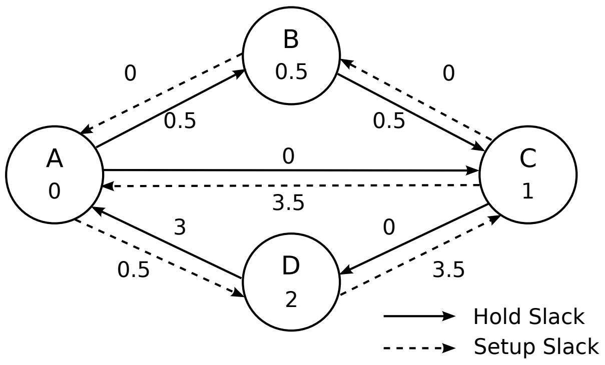


Figure 2.8: Slack graph with Bellman-Ford distances for period 3.5. From the constraint graph of Figure 2.7.

2.9 Distributed Slack Graph

The slack of an edge gives the criticality of that edge depending on the assigned skews of its source and destination registers. This slack depends on how the skews where assigned and it cannot be used to determine the real criticality of that edge on a global context. However, it is possible to distribute this slack around the cycle in a way that all edges get the corresponding global slack.

A **Distributed Slack Graph** is an Slack Graph where for each node the minimum input slack is the same as the minimum output slack. The skew assignment that has this property is unique. This problem is also called *minimum balanced* problem and Young,Tarjan,Orlin [27] solve it using a parametric shortest path algorithm that can be used to calculate the *maximum mean cycle* and the *minimum balanced graph*. This new slacks can be assigned to registers and become a node property instead of an edge property that measures criticality of the node.

A simple algorithm that distribute pairs of nodes and iterates until there is no more unbalanced pairs has been tested and performs very well for big circuits. The fact that all edges have a similar delay by design makes this algorithm to perform few iterations.

Figure 2.9 shows the example with a period of 3.5 with distributed slacks. Nodes (Table 2.1) and edges (Table 2.2) can be sorted by criticality. This information can be used by engineers to optimise the circuit at these points.

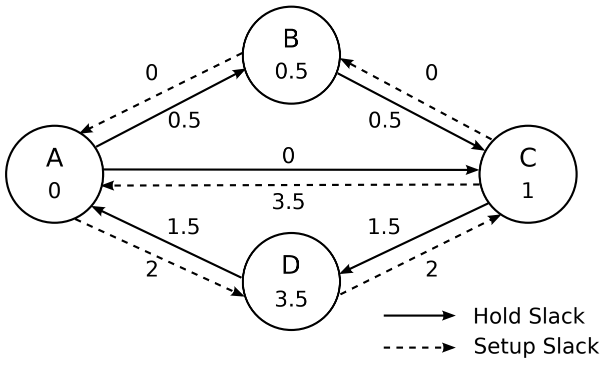


Figure 2.9: Distributed slack graph for period 3.5

Node	Distributed Slack
A	0
B	0
C	0
D	1.5

Table 2.1: Top critical nodes by distributed Slack

Edge	Distributed Slack
$A \rightarrow C$	0
$C \rightarrow B$	0
$B \rightarrow A$	0
$A \rightarrow B$	0.5
$B \rightarrow C$	0.5
$C \rightarrow D$	1.5
$D \rightarrow A$	1.5
$A \rightarrow D$	2
$D \rightarrow C$	2
$C \rightarrow A$	3.5

Table 2.2: Top critical edges by distributed slack

Chapter 3

Clustering algorithms

This chapter presents two algorithms to cluster registers: the algorithm of Section 3.1 minimises the number of connections of the cluster graph, and the algorithm of Section 3.2 minimises the global period and the cluster placement.

3.1 Local Search to minimise the number of connections

The local search algorithm to minimise number of connections of the cluster graph starts from an initial clustering configuration and iteratively tries to move nodes from one cluster to any other that increase the global cost function. Not all movements are valid, since there are minimum and maximum size constraints on the clusters. It is not possible to move a node into a cluster which already has the maximum size, and it is not possible to move a node out of a cluster which has the minimum size. The initial clustering configuration is based on a topological order. The algorithm will try to move nodes until there is no node that can increase the cost function.

3.1.1 Cost Function

A Cost function gives a numeric quantification of how good a cluster configuration is. Having a register graph G and a Cluster graph CG, the set of register edges (links) between two clusters and the total cost are defined as:

$$links(c_1, c_2) = (u, v) \in G \mid u \in c_1 \wedge v \in c_2$$
$$TotalCost = \sum_{c_1, c_2 \in CG} \frac{1}{\mid links(c_1, c_2) \mid +1}$$

The cost function $TotalCost$ is a multiplicative inverse function. It is designed to reward movements that lead to a solution with few cluster edges by decreasing the edges with less links. It is a better to decrease an edge with fewer links (the solution is closer to 0 links) than and edge with much more links (clusters will be connected anyway). For example, it is better to decrease the number of links from 5 to 4 than from 50 to 49. The local search is going to the solution where the edge with 5 to 4 links will become 0.

The $TotalCost$ function has a *range* of values depending on the register graph and the number of clusters. The maximum and minimum cost are studied by constructing the best and worst case as follows:

Maximum total cost

For N nodes in C clusters the maximum cost of the function is from a clustering with no links between clusters. Two different expressions are shown depending on if self loops are minimised or ignored. The cost of not being connected to all clusters is the total number of possible edges C^2 (or without self loops: $C \times (C-1)$) times the cost of the edge $\frac{1}{1}$.

$$MaximumCost_{self\ loops} = C^2 \times \frac{1}{1} = C^2$$

$$MaximumCost_{no\ self\ loops} = C \times (C-1) \times \frac{1}{1} = C^2 - C$$

Minimum total cost

For N nodes in C balanced clusters the minimum cost of the function is from a clustering with all possible edges between clusters. Two different expressions are shown depending on if self loops are minimised or ignored. A complete register graph with $\frac{N}{C}$ links between clusters has a total cost of the number of edges C^2 (or without self loops: $C \times (C-1)$) times the cost of each edge $\frac{1}{\frac{N}{C}+1}$.

$$MinimumCost_{self\ loops} = C^2 \times \frac{1}{\frac{N}{C}+1} = \frac{C^2}{\frac{N}{C}+1}$$

$$MinimumCost_{no\ self\ loops} = C \times (C-1) \times \frac{1}{\frac{N}{C}+1} = \frac{C^2 - C}{\frac{N}{C}+1}$$

3.1.2 Cluster graph adjacency matrix

A function that calculates the cost function of a cluster edge needs to know the number of links of that edge. It needs to count how many register edges are between the clusters. As the number of clusters is expected to be small, it is affordable to store C^2 costs. A cluster graph adjacency matrix is a $C \times C$ matrix storing the number of links for each cluster edge. This data structure is used by the algorithm in Section 3.1 to have constant access to the number of links for a given cluster edge, and to update the cost of a cluster edge incrementally when moving one node from one cluster to another.

3.1.3 Local Search algorithm

The LocalSearch Algorithm 4 tries to move one node at each iteration until no node can be moved. The key of the algorithm is at the computation of the increment on the cost function when moving one node needs to be cheap. The same node can be moved several times because the configuration is constantly changing. Nodes need to be visited again every time the configuration has been changed because this change can make the movement possible.

Algorithm 4 LocalSearch

```
Require: $list$ is a List of all nodes; all nodes are assigned to clusters.
Ensure: There is no node movement that can improve the cost function.
1: while $noimprove\_counter < list.length()$ do
2:   $node \leftarrow list.next()$
3:   $improvement \leftarrow Move(node)$ {Tries to move $node$ to another cluster, improving the cost function.}
4:   if $improvement > 0$ then
5:     $noimprove\_counter \leftarrow 0$
6:   else
7:     $noimprove\_counter \leftarrow noimprove\_counter + 1$
8:   end if
9: end while
```

A node can be moved (Algorithm 5) to another cluster if the origin cluster is bigger than the minimum cluster size and the destination cluster is smaller than the maximum cluster size. Moreover, a node is moved if it increase the total cost. The $Move$ function calculates the cost of the matrix before moving the node, and the cost after moving it to all possible destinations. The $MoveNode$ function change the assignation of the node to the new cluster and only needs to go through all predecessors and successors to calculate how the *links* will be affected on the adjacency matrix. Thanks to this matrix there is no need to go through all nodes and edges of the register graph to calculate the $TotalCost$ value. At the end it moves the node to the best destination, if it increase the cost.

3.1.4 Example

This section shows an example of one iteration of the LocalSearch algorithm. Having an initial clustering configuration of 9 nodes and 3 clusters as in Figure 3.1, a cost increment of moving the node u to cluster C or B will be calculated.

The range of TotalCost (ignoring self loops) in this example will be:

Algorithm 5 Move

Require: $node$ is the node to be moved;$current\_cluster$ is the current cluster of $node$; $configuration$ is the current clustering solution.

Ensure: The $node$ is moved to a better cluster configuration with higher cost value; $configuration$ is updated according to the movement; returns the cost increment.

```
1: $current\_cost \leftarrow TotalCost(configuration)$
2: $increment \leftarrow 0$
3: **if** $current\_cluster.size() > MIN\_SIZE$ **then**
4:   **for all** $cluster$ **do**
5:     **if** $cluster.size() < MAX\_SIZE$ **then**
6:       $new\_configuration \leftarrow MoveNode(node, cluster, configuration)$ {Moves the $node$ into cluster $cluster$ for a cluster $configuration$, returns the new configuration }
7:       $cost \leftarrow TotalCost(new\_configuration)$ {$MoveNode$ can calculate the increment of cost at the same time that is generating the new configuration.}
8:       **if** $cost > current\_cost$ **then**
9:         $increment \leftarrow increment + cost - current\_cost$
10:        $current\_cost \leftarrow cost$
11:        $configuration \leftarrow new\_configuration$
12:      **end if**
13:    **end if**
14:  **end for**
15: **end if**
16: return $increment$
```

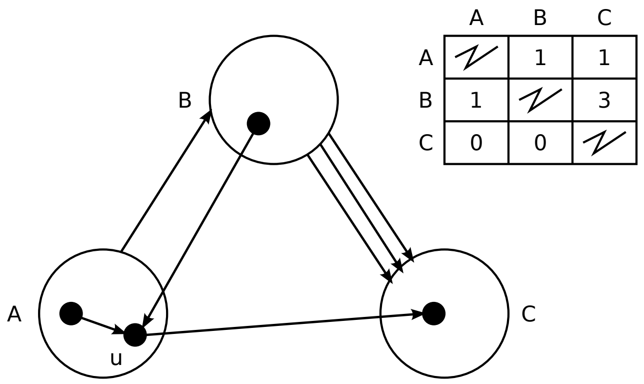
- Minimum : $\dfrac{C^2 - C}{\frac{N}{C} + 1} = \dfrac{9 - 3}{\frac{9}{3} + 1} = 1.5$
- Maximum : $C^2 - C = 6$

The cost of a given configuration is calculated from the adjacency matrix (in this example self loops are ignored). Notice how to update the matrix when a node is moved depends only on the predecessors and successors of that node. The initial configuration cost and the improvements of moving the node u are the following:

Initial Configuration Figure 3.1 has a $TotalCost = \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{1} + \frac{1}{1} = 3.75$.

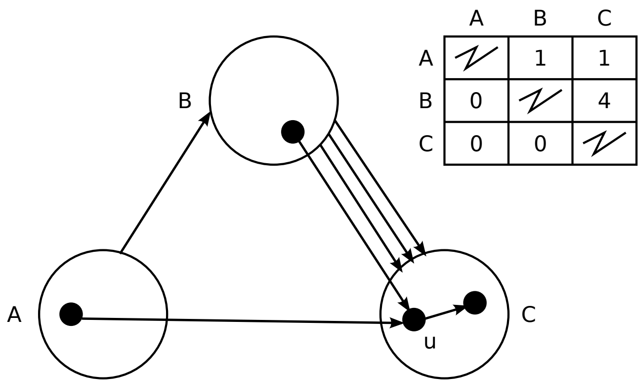
Moving u to C Figure 3.2 has a $TotalCost = \frac{1}{2} + \frac{1}{2} + \frac{1}{1} + \frac{1}{5} + \frac{1}{1} + \frac{1}{1} = 4.20$. It increments the cost with 0.45.

Moving u to B Figure 3.3 has a $TotalCost = \frac{1}{3} + \frac{1}{1} + \frac{1}{1} + \frac{1}{5} + \frac{1}{1} + \frac{1}{1} = 4.53$. It increments the cost with 0.78. The node u will be moved into cluster B because it maximises the cost more than moving it to C.



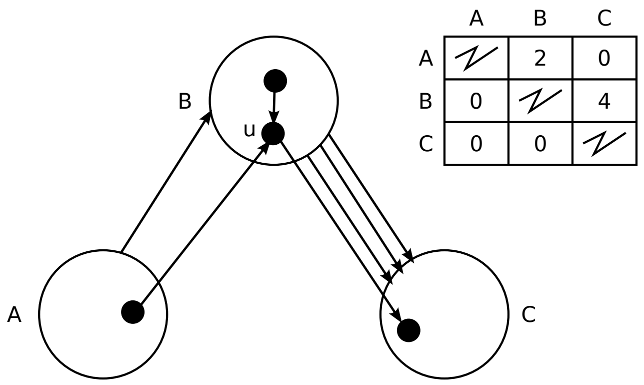
	A	B	C
A		1	1
B	1		3
C	0	0	

Figure 3.1: Cluster configuration with node u in cluster A



	A	B	C
A		1	1
B	0		4
C	0	0	

Figure 3.2: Cluster configuration with node u in cluster C



	A	B	C
A		2	0
B	0		4
C	0	0	

Figure 3.3: Cluster configuration with node u in cluster B

3.2 K-means by position and skew

The algorithm of this section faces the clustering tradeoff between placement and performance. It is important to have compact clusters to minimise the total cost of distributing the clock signal to registers, as well as other benefits in terms of variability. Performance of the clustered solution is obviously also important. To be able to cope with the placement vs performance tradeoff the algorithm uses a parameter to tune the balance.

The algorithm uses *K-means* [16, 17, 18] to group registers that are closer in distance. For placement, Manhattan distances gives a better approximation of the final routing cost than euclidean distances because of the physical grid implementation. From the performance point of view the distance of skews is used as an heuristic to optimise the period. The heuristic comes from the idea that two registers with a similar skew, when grouped together, the lost of performance will be minimum. When grouping two registers with diffrent skew, the maximum period that could be lost is the difference of skews.

If two registers i and j with skews X_i and X_j are grouped together implies that both will share the same skew and $X_i = X_j$.

No Hold Constraint can be violated when two registers are grouped because the minimum delay is always greather than 0.

$$\begin{aligned} Hold\,Constraint: MIN_{ij} &\geq X_j - X_i & before \\ MIN_{ij} &\geq 0 & after \end{aligned}$$

Let P_1 be the period before grouping the 2 registers, and P_2 the period after grouping them. P_2 can be increased as much as $(X_j - X_i)$ compared to P_1.

$$\begin{aligned} Setup\,Constraint: MAX_{ij} &\leq P_1 + (X_j - X_i) & before \\ MAX_{ij} &\leq P_2 + 0 & after \\ P_1 \leq P_2 &\leq P_1 + (X_j - X_i) \end{aligned}$$

A global distance function is defined by composing the placement distance plus the skew distance using α and β parameters to balance them. Forming a 3D space (skew-placement) for *K-means*.

$$\begin{aligned} Distance(r_1, r_2) =& \alpha \times DistancePlacement(r_1, r_2) + \\ & \beta \times DistanceSkew(r_1, r_2) \\ DistancePlacement(r_1, r_2) =& abs(r_1.x - r_2.x) + abs(r_1.y - r_2.y) \\ DistanceSkew(r_1, r_2) =& abs(r_1.skew - r_2.skew) \end{aligned}$$

The coordinates on the layout are provided by a placement algorithm, but there are many possible valid solutions to assign skews. It is possible to minimise the total skew [11], or distribute the slacks to classify registers and edges by criticality like in Chapter 2 Section 2.9 which sets skews far from the interval extremes (more robust to timing errors).

One good property that can be used in combination with placement is to have all non-critical registers in one big cluster, and small clusters for the critical registers to improve the period. After this clustering the big none-critical cluster can be re-clustered by placement without affecting performance.

The used skews are calculated by setting 0 to all registers and applying Bellman-Ford on the Slack Graph. Bellman-Ford will update just the registers that have a negative slack edge (a shortcut) and keep skew 0 to the none critical ones.

3.2.1 K-means

The K-means algorithm creates initial centroids spared on the skew-placement space. Each iteration of the algorithm assigns each node to the closest centroid, and it recalculates the new positions of the centroids as the mean position for each dimension (x, y, skew). K-means stops iterating when no node jumps from one centroid to another. On the Results Section 4.4.2 (Page 47) there are some figures that show the dimensions on a 3D plot.

3.2.2 Evaluation

To evaluate a clustering solution needs to be done from the performance point of view and from the placement point of view. The evaluation of the clustering in terms of performance is simply the period of the Cluster Graph. Described in Chapter 2. The evaluation in terms of placement is done by calculating spanning trees using Prim's algorithm [20].

Prim's algorithm starts from a node and builds the spanning tree by adding the closest node to the tree at each iteration. This algorithm has a computational cost of $\Theta(n^2)$ because the distance graph is complete.

Spanning trees can be hard to calculate for huge amount of nodes. For big circuits a “fast” option has been implemented that simplifies the space by dividing the area

Algorithm 6 K-means

```
 1: $movement \leftarrow true$
 2: while $movement$ do
 3:     $movement \leftarrow false$
 4:     for all register $r$ do
 5:         $new\_cluster \leftarrow SearchClosestCluster(r)$
 6:         if $new\_cluster \neq r.cluster$ then
 7:             $r.cluster \leftarrow new\_cluster$
 8:             $movement \leftarrow true$
 9:         end if
10:     end for
11:     $RecalculateCentroids()$
12: end while
```

into a grid and calculates the spanning tree of each cell area plus the global spanning tree using representants of each cell to connect all nodes. The computational cost remains the same $\Theta(n^2)$ but it is divided by the size of the grid.

Chapter 4

Results

This Chapter presents some results of executing the two algorithms of Chapter 3. The first section describes the execution environment, the second runs the algorithm to minimise the number of connections, the third study the performance using ISCAS benchmark and OpenSparc, the forth section uses the algorithm that combines performance with placement to study the tradeoff. Some illustrative plots are presented to show the skew-placement space of OpenSparc design.

4.1 Execution environment

The algorithms were implemented on the de-synchronisation flow of *Elastix* EDA tool. The synthesis and placement of the designs have been obtained from an industrial tool.

Elastix EDA tool with both algorithms have been executed on an Intel Core Duo 2 processor at 2.13GHz with 4MB cache and 4GB of RAM running linux 2.6.23.

ISCAS benchmark and OpenSparc are the tested designs in this section. The OpenSparc design from *Sun Microsystems* is an open source multi-processor and multi-thread design. A single processor has been used on these experiments which has: 190 inputs, 132 outputs, 15527 registers, and 480325 paths.

4.2 Number of connections

The LocalSearch algorithm presented on Chapter 3 Section 3.1 has been tested on the practical design OpenSparc because of its big size. Table 4.1 and the corresponding plot on Figure 4.1 show the results of 10 register clusters. Each execution for 10 clusters toke about 10 seconds.

The worst case scenario for 10 register clusters, plus 1 cluster for inputs, and 1 cluster for outputs is: 10 connections from inputs to register clusters, 10 from registers to output, and a complete connected graph of registers 10^2. This worst case gives a 120 connections. The best case scenario is a pipe line with 11 edges.

The relation between nodes (15849) and edges (480325) is about 30 edges per node. This big relation could mean that if the clustering is not aware of connections could easily generate a strongly connected cluster graph. For example a clustering of the k-means algorithm based on placement gives 113 connections, and based on performance 110. These two clustering examples prove how close to the worst case scenario a clustering algorithm can be without taking into account the connections.

Figure 4.1 shows the evolution of the algorithm when increasing the balance factor. The balance factor allows the algorithm to build clusters of different sizes. A balance factor of 0.2 means that a cluster can be 0.2 times bigger or smaller than the mean size.The plot uses the standard deviation as a real measure instead of the balance factor that just provides limits on the cluster sizes. Notice how the evolution is not monotonic: the solution with balance 0.5 is worst than the solution of 0.4. This behaviour happens because the algorithm implements a heuristic and the optimal solution is not guaranteed. The results can be improved by running the algorithm several times adding some in-determinism and changing the initial solution (currently based on topological order).

Balance	#edges	std
0.001	108	5.100
0.1	93	152.491
0.2	87	302.920
0.3	86	406.967
0.4	84	573.358
0.5	85	693.251
0.6	79	800.991
0.7	77	905.689
0.8	78	1072.582
0.9	74	1157.232
1.0	64	1258.712
1.5	59	1579.163
2.0	48	1904.196

Table 4.1: Number of edges on the cluster graph. For 10 cluster solution on OpenSparc.

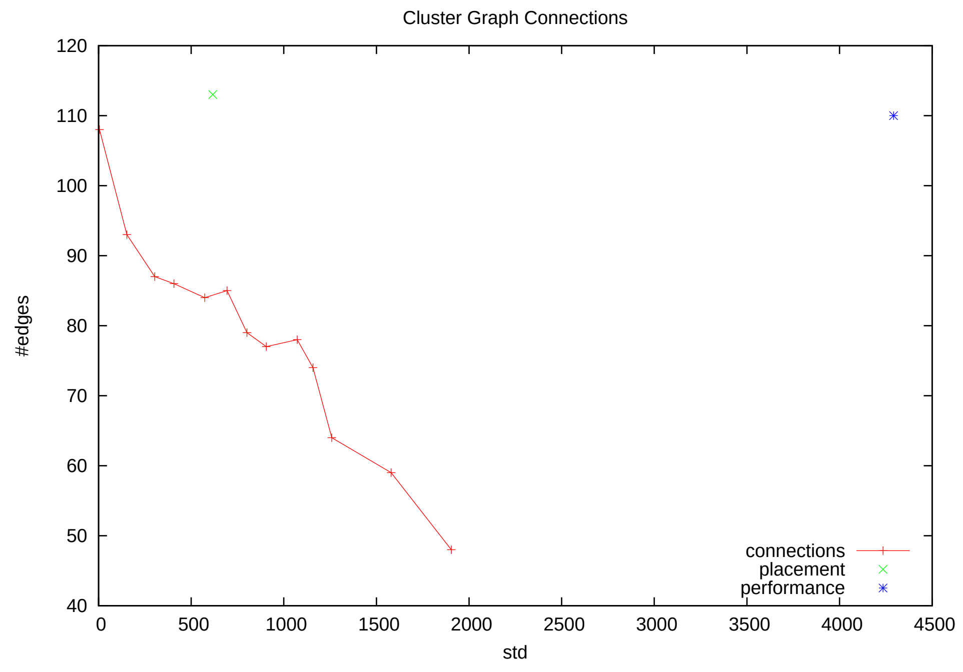


Figure 4.1: Local search solution minimising number of connections for 10 clusters on Sparc. Showing the solution based on placement, and the solution based on performance.

4.3 Performance

This section studies how the algorithm behave in terms of performance. Optimum periods are calculated with several cluster solutions using ISCAS benchmarks and OpenSparc circuit.

4.3.1 ISCAS

To evaluate the performance of ISCAS benchmark the k-means algorithm with 100% on skew has been run it. Table 4.2 and 4.3 show some of ISCAS circuits. These tables show the name of the circuit, the total number of nodes (including inputs outputs and registers), the total number of edges, and the optimum period. Table 4.2 shows the values for Hold&Setup period and Table 4.3 for just Setup period. The period with Hold&Setup constraints is calculated by the binary search (Chapter 2 Section 2.7), and the period with only Setup Constraints is given by the maximum mean cycle (Chapter 2 Section 2.3). The optimum period (P_opt) is the period of having as many clusters as registers has the circuit, plus one cluster for inputs and another for outputs.

These tables compare 1,3, and 5 cluster solutions with the optimum period. The 1-cluster solution represents the period of the circuit without appling clock skew optimisation because all registers are synchronised (have the same clock signal). The periods are printed using normalised values to appreciate the relative value of the period compared to the optimum one.

Some important results are described on the next items:

- Both tables represent the same clusters but the skews used by k-means algorithm have been calculated by using just Setup Constraints. The algorithm will behave better on optimising Setup period than Setup&Hold period, but it can not be conclude that the average values for Setup period are better than the values of Setup&Hold just because of this. Adding Hold constraints change completely the instance of the problem.
- Some circuits are acyclic ($s1196$ and $s1238$) and no setup period exist because there are no cycles. The period in this case can be defined as 0 because it is possible to assign infinite skews to registers without braking any setup constraint. However, there is always a Hold&Setup period because these two constraints are creating cycles by construction.
- The Setup Period is always smaller than the Hold&Setup because the second has all the extra Hold constraints to satisfy. Hold&Setup Period can be improved up to the Setup Period by fixing all violations on Hold constraints. This can be done by adding delays on some *min delay* paths. The Setup period is the reference to use because it can be achieved.
- It is important to compare the optimum period (P_opt) with the worst one (1 cluster) to understand the possible margin that is possible to improve by using clock skew. Notice that the proportion between them is just 1.223 (22.3%)

for Hold&Setup and 1.291 (29.1%) for Setup. This difference represents the maximum gain achievable by using clock skew. Notice how circuit $s35932$ (Table 4.3) has the same Setup Optimum Period and 1-cluster period, which means that the maximum delay is on the critical cycle and it can not borrow time from any other register.

- The expected improvement from 1 cluster to 3 and 5 clusters can be easily seen with the average values. For Setup&Hold Period: starting from the worst case of 1-cluster with 1.223 of the optimal, can be improved to 1.120 with 3 clusters and to 1.105 with 5 clusters. This means that on average, the 5-cluster solution is at 10.5% closer to the optimum period. For just Setup Period: 1-cluster is at 1.291 of the optimal, 3 clusters at 1.097 and 5 clusters at 1.042 (4.2% of the optimum period). A closer evolution of the period by increasing the number of clusters is studied with the OpenSparc circuit on Section 4.3.2.
- The best results with Setup period and 3 clusters are obtained on circuits: $s35932$ (100% even for 1 cluster), $s510$, and $s832$. The circuits $s641$ and $s713$ also achieve the optimal period with 5 clusters.
- The period of 5 clusters is always better than the period for 3 clusters. It is desirable that the period decreases monotonically when the number of clusters increases. This behaviour is not guaranteed by the algorithm as it can be seen on the next section with OpenSparc.
- All executions are really fast. For example, the execution time for clustering the circuit $s38417$ toke only 0.822s for 3 clusters and 0.74s for 5 clusters.

4.3.2 OpenSparc

To evaluate the evolution of the period by increasing the number of clusters, the k-means algorithm with 100% skew has been run it on OpenSparc. The Table 4.4 shows the period from the worst case solution (1 register cluster, 1 input cluster, and 1 output cluster) to the best possible period (as many clusters as registers). Figure 4.2 shows how the progression is not monotonic. For example, the heuristic is giving better period for 3 clusters than for 4, and it is possible to get same period as 3 clusters just by splitting one of them.

The execution time to clusterise is small compared to the time to calculate the skews. For example, the 3 cluster solution toke 25.7s to find the optimal period and calculate the skews, and 7.26s to clusterise it.

Circuit				Normalised Setup&Hold Period			
name	nodes	edges	P_opt	P_opt	1 cluster	3 clusters	5 clusters
s1196	46	387	1.571	1	1.1814	1.1222	1.1222
s1238	46	379	1.332	1	1.1096	1.0901	1.0901
s13207	429	1445	0.946	1	1.1776	1.0264	1.0148
s1423	96	2235	2.193	1	1.2312	1.0461	1.0461
s15850	203	919	0.997	1	1.3159	1.0602	1.0251
s298	23	86	0.706	1	1.3994	1.1331	1.0822
s35932	1795	7051	1.832	1	1.2604	1.2598	1.2587
s38417	1698	33731	2.335	1	1.1375	1.0385	1.0385
s38584	1410	12058	1.545	1	1.3631	1.3003	1.2867
s386	20	129	0.828	1	1.1594	1.0060	1.0060
s400	30	175	0.725	1	1.4166	1.1931	1.1931
s444	30	175	0.779	1	1.1964	1.0205	1.0077
s510	32	103	1.023	1	1.0137	1.0000	1.0000
s526	30	167	0.828	1	1.1969	1.1316	1.0000
s5378	244	2180	1.096	1	1.4772	1.4352	1.4142
s641	77	486	1.120	1	1.2571	1.2500	1.2500
s713	77	486	1.151	1	1.1911	1.1616	1.1616
s820	42	213	1.164	1	1.0713	1.0043	1.0043
s832	42	213	1.186	1	1.0902	1.0000	1.0000
Average:				1	1.223	1.120	1.105

Table 4.2: ISCAS benchmark with periods for Setup&Hold constraints. Showing cluster variations of 1, 3, and 5 clusters compared to the optimum period.

Circuit				Normalised Setup Period			
name	nodes	edges	P_opt	P_opt	1 cluster	3 clusters	5 clusters
s1196	46	387	-	-	-	-	-
s1238	46	379	-	-	-	-	-
s13207	429	1445	0.946	1	1.1776	1.0264	1.0148
s1423	96	2235	2.008	1	1.3446	1.0478	1.0478
s15850	203	919	0.997	1	1.3159	1.0602	1.0251
s298	23	86	0.706	1	1.3994	1.1331	1.0113
s35932	1795	7051	1.259	1	1.0000	1.0000	1.0000
s38417	1698	33731	2.335	1	1.1375	1.0210	1.0158
s38584	1410	12058	0.979	1	2.0031	1.4188	1.2462
s386	20	129	0.827	1	1.1608	1.0073	1.0073
s400	30	175	0.694	1	1.4798	1.1383	1.0605
s444	30	175	0.752	1	1.2394	1.0519	1.0332
s510	32	103	1.023	1	1.0137	1.0000	1.0000
s526	30	167	0.694	1	1.4280	1.3501	1.1268
s5378	244	2180	1.021	1	1.5857	1.2968	1.1175
s641	77	486	1.059	1	1.2937	1.0085	1.0000
s713	77	486	1.144	1	1.1984	1.0804	1.0000
s820	42	213	1.164	1	1.0713	1.0034	1.0034
s832	42	213	1.186	1	1.0902	1.0000	1.0000
Average:				1	1.291	1.097	1.042

Table 4.3: ISCAS benchmark with periods for Setup constraints. Showing cluster variations of 1, 3, and 5 clusters compared to the optimum period.

Clusters	Setup Period (ns)	% to P_opt
1	1.659	81.01
2	1.603	83.84
3	1.451	92.63
4	1.471	91.37
5	1.421	94.58
6	1.409	95.39
7	1.390	96.69
8	1.380	97.39
9	1.380	97.39
10	1.371	98.03
11	1.377	97.60
12	1.369	98.17
14	1.371	98.03
15	1.363	98.61
16	1.364	98.53
17	1.361	98.75
19	1.358	98.97
P_opt	1.344	100.00

Table 4.4: Sparc Performance depending on the number of clusters

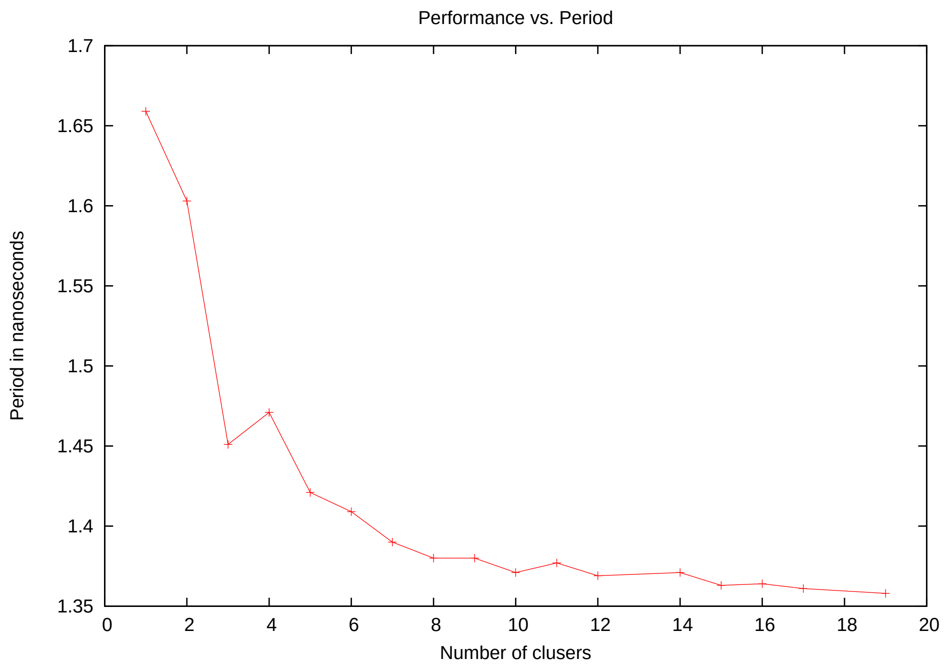


Figure 4.2: Sparc clusters vs Period. Optimum period: 1.344, Optimum period with hold constraints: 1.565, Max Delay: 1.659

4.4 Performance vs Placement

This section studies the relation between placement and performance. OpenSparc has been choose because of its size (compared to ISCAS benchmark). First, a 5-cluster solution is explored in detail by balancing placement vs performance, and after some plots of the 3D space is showed to analyse how the k-means works.

4.4.1 A 5-cluster solution for OpenSparc

The k-means solution combining placement and performance is explored on OpenSparc design. Table 4.5 shows the Setup period and the spanning tree cost evolution from the 100% skew and 0% placement to the 0% skew and 100% placement. The period is given by the maximum mean cycle with Howard's algorithm (Chapter 2 Section 2.4) and the Spanning Tree column represents the sum of spanning trees of each cluster (using Prim's algorithm). Figure 4.3 shows the evolution of the period and Figure 4.4 shows it for placement. Figure 4.5 plots the trade-off between having a good clustering in terms of performance or in terms of placement. The engineer can choose the solution that fits better the requirements. The solution with 40% skew and 60%placement appears to be a good balanced solution. Notice how extreme solutions behave irregular because of the noise from the other parameter (the small percentage noise).

Skew %	Placement %	Period (ns)	Spanning Tree (μm)
100	0	1.421	107070
90	10	1.421	106951
80	20	1.413	106751
70	30	1.413	106614
60	40	1.411	106344
50	50	1.471	103587
40	60	1.471	100282
30	70	1.659	94746
20	80	1.659	93239
10	90	1.659	93020
0	100	1.648	92972

Table 4.5: Sparc 5-cluster Performance vs Placement: Balance percentage, Setup Period, and sum of all 5 spanning trees.

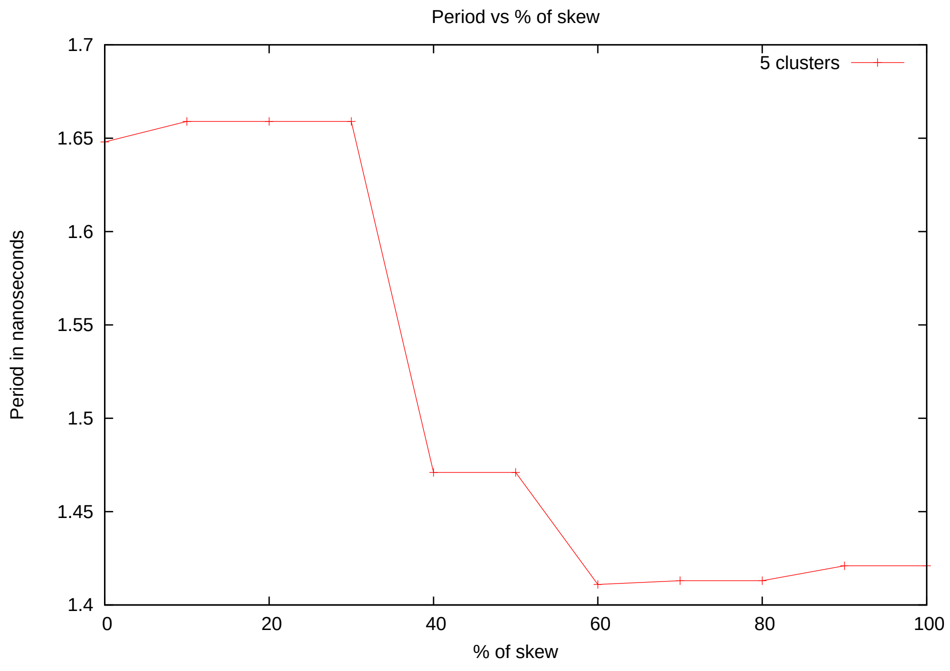


Figure 4.3: Sparc with 5 clusters. Using K-means balancing skew and placement by increments of 10%

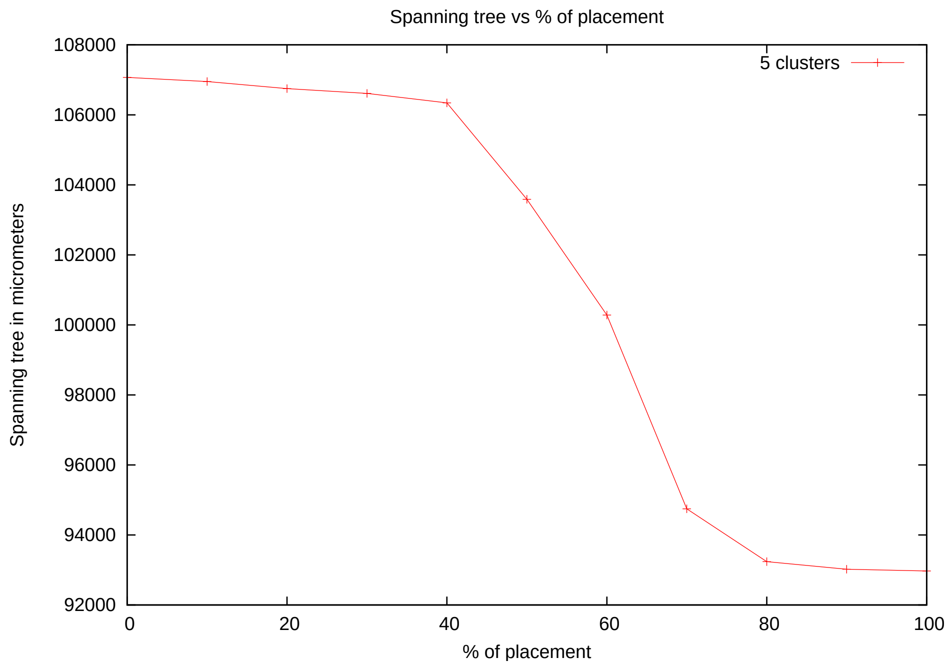


Figure 4.4: Sparc with 5 clusters. Using K-means balancing skew and placement by increments of 10%

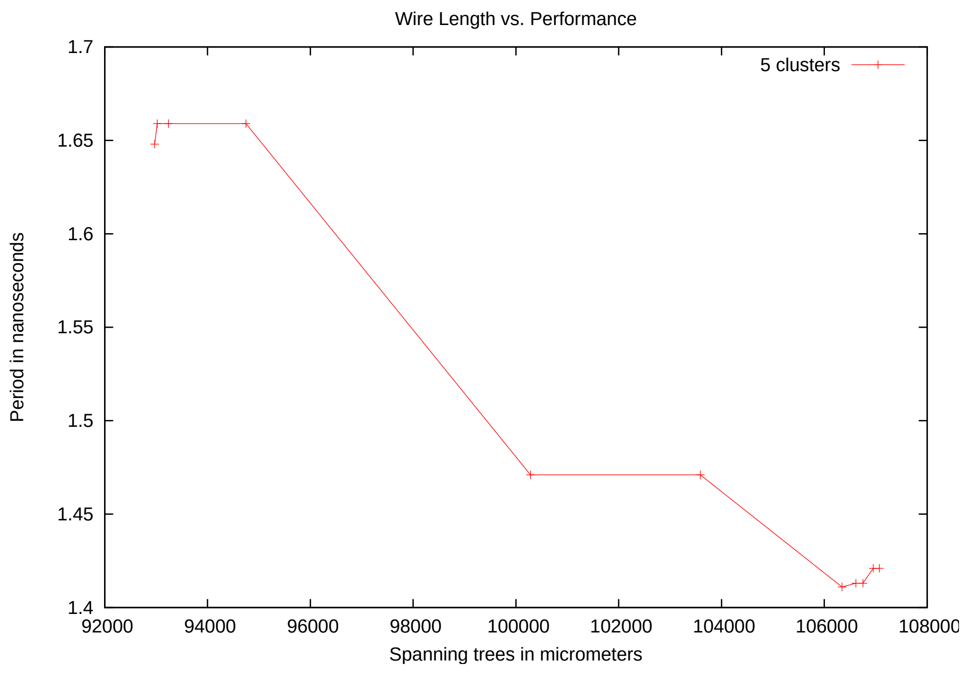


Figure 4.5: Sparc with 5 clusters. Total wire length vs Period

4.4.2 OpenSparc skew-placement space

To show how k-means is grouping by distance on the skew-placement space the OpenSparc has been used with 5 clusters.

Figures 4.6 , 4.7 , 4.8, 4.9, 4.10, and 4.11, are a list of plots to illustrate 3 solutions. The 2d and 3d space of solutions with 0%-100%, 100%-0%, and 50%-50% are printed.

100% placement Figure 4.7 shows how the area is perfectly partitioned into 5 clusters and Figure 4.6 shows how skews are grouped by columns. This solution gives the worst possible period because it is grouping nodes that need some skew difference to get the optimal period.

100% skew Figure 4.8 shows how clusters are based on layers of the skew dimension. It produce one big cluster for skew 0 and other small clusters for different levels. $regs1$ is the big cluster that can be re-clustered by placement if desired. The skew of the cluster is annotated on the right side of the plot and shows the common skew for all nodes of the cluster. Figure 4.9 shows how $regs2$ and $regs3$ are spared around all the surface and overlapping their areas. The nodes of cluster $regs4$ are not badly grouped from the point of view of the spanning tree because nodes are spared but in compact blocs. Finally, the cluster $regs5$ is compact on all dimensions.

50% - 50% Figure 4.10 compared to Figure 4.8 shows how giving 50% to placement the k-means algorithm is able to fix clusters $regs2$ and $regs3$ by grouping them in terms of placement. Cluster skews are annotated on the right side showing small differences comparing with the 100% skew solution. Changing the skew of a cluster pulls other clusters as it happens for $regs4$ and $regs5$ when $regs2$ and $regs3$ skews are reduced.

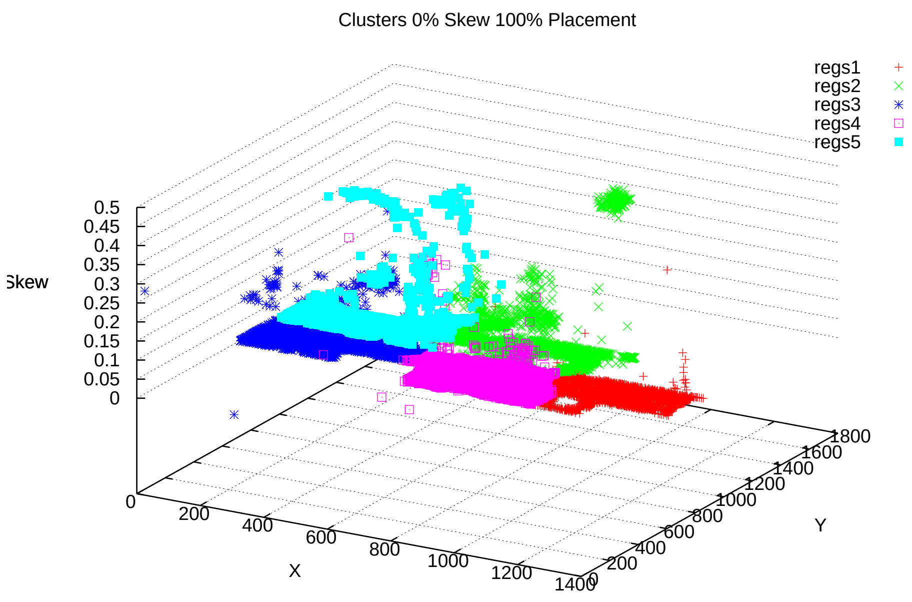


Figure 4.6: 3D Sparc with 5 clusters. K-means with 100% placement

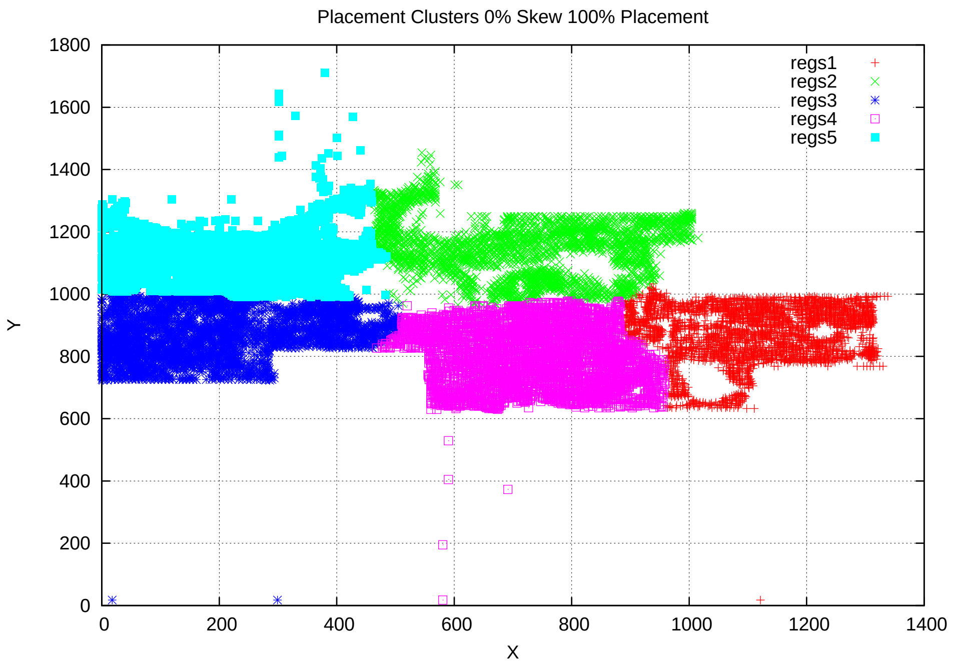


Figure 4.7: 2D Sparc with 5 clusters. K-means with 100% placement

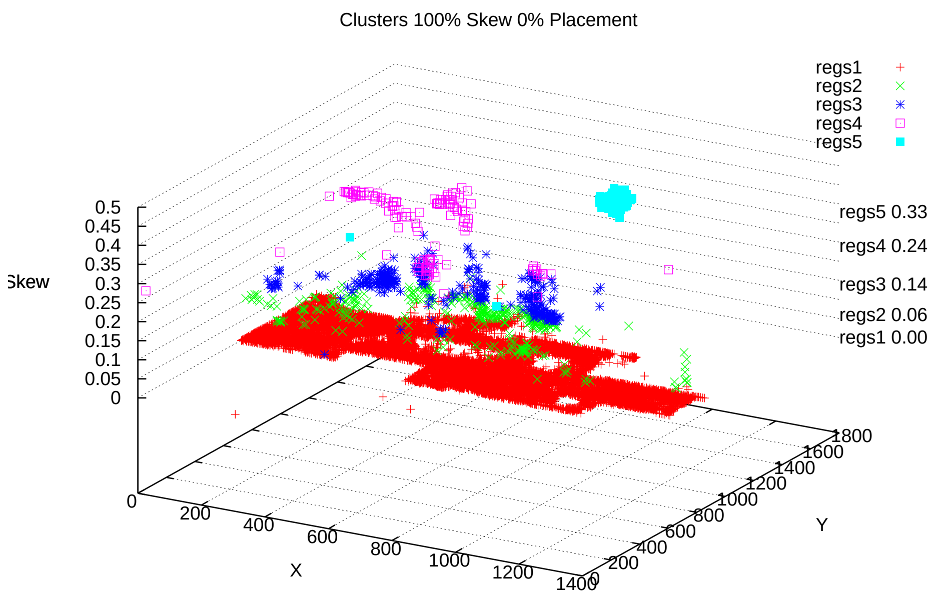


Figure 4.8: 3D Sparc with 5 clusters. K-means with 100% skew

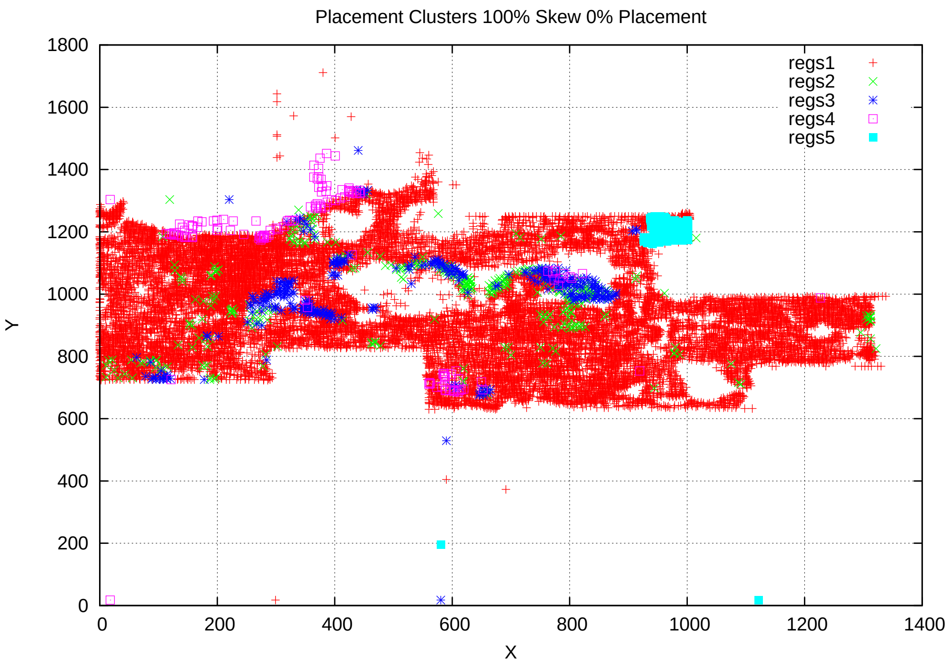


Figure 4.9: 2D Sparc with 5 clusters. K-means with 100% skew

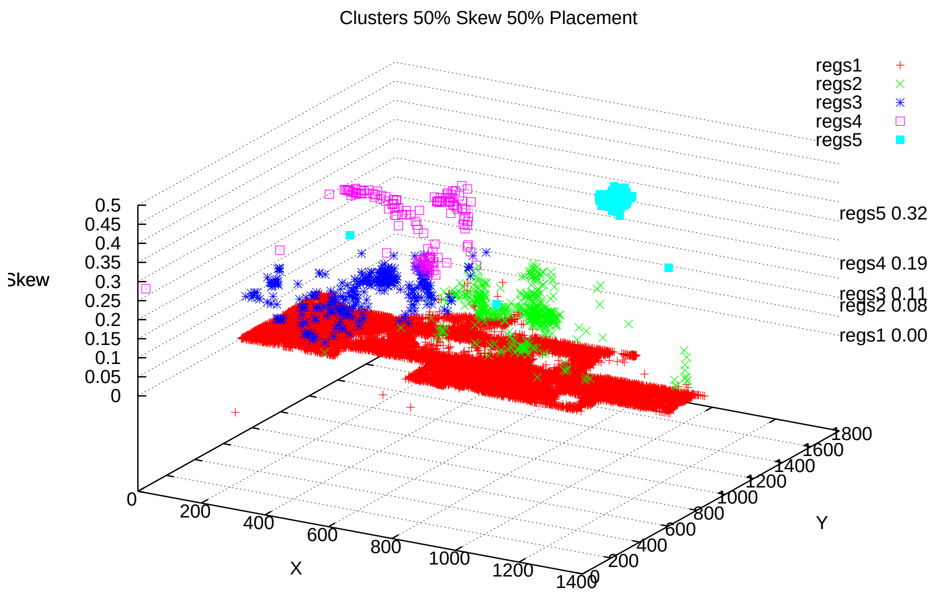


Figure 4.10: 3D Sparc with 5 clusters. K-means with 50% skew 50% placement

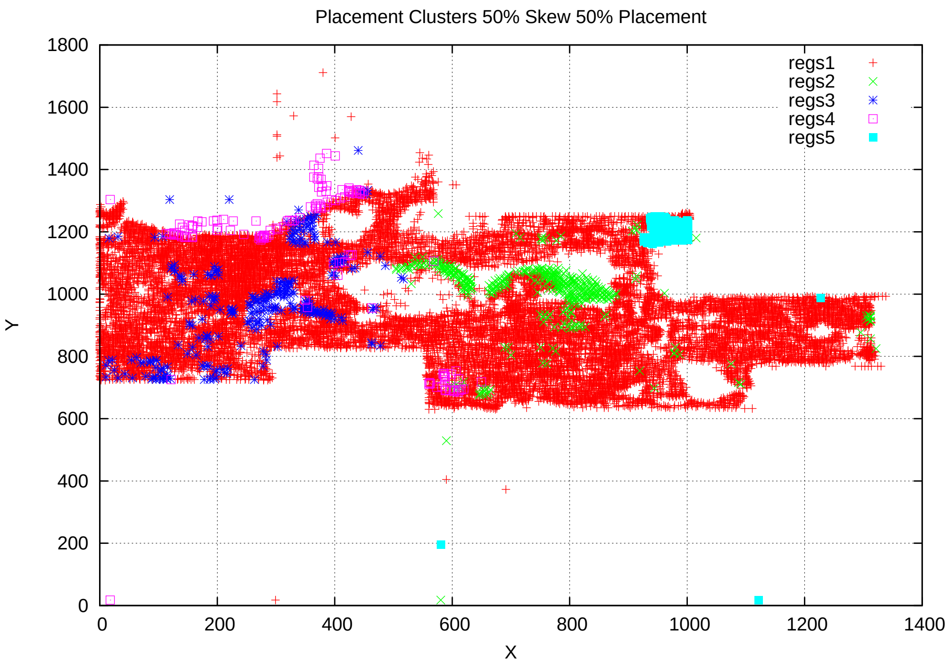


Figure 4.11: 2D Sparc with 5 clusters. K-means with 50% skew 50% placement

Chapter 5

Conclusions

Optimisation algorithms for clustering of circuit registers belongs to the heuristic world. The size of circuits is too big for algorithms that behave slower than linear. Moreover, algorithms needs to optimise several tradeoff properties like performance, placement, connections, etc. The presented algorithm for connections minimisation gives good results and it can be easily extended to optimise other properties. The K-means algorithm can provide good performance combined with a good placement. Both algorithms can be used incrementally. It is possible to re-cluster a subset of nodes. For example the engineer can start with an initial cluster based on performance and recluster the non critical clusters based on connections or placemet to minimise other properties as area or wire length.

The binary search to find the optimal period and the two clustering algorithms have been implemented on *Elastix Corporation* EDA tool. These algorithms are currently used on the *De-synchronisation* flow of that tool. Algorithms have been tested with public ISCAS benchmark and OpenSparc processor on the Results section.

Future Work

Some improvements can be done to the current local search algorithm. It is possible to add layout positions, skew, and slack to the cost function and be able to combine connections, placement and performance tradeoff on the same algorithm. The idea of combining everything can result into an algorithm that gives bad results for each characteristic, but it needs to be explored. The current local search is fast and it can be improved by converting it into an Iterative Local Search Algorithm, adding some perturbations to be able to find a better solution.

K-means algorithm is also very fast, some computation can be added to improve the quality:

- The distance function can be improved by adding the distributed slack information to know how flexible is a register.

- Refine the process by reassigning skews to registers according to the current cluster solution and checking if any register is better placed in another cluster.
- The critical cycles are known since the beginning and they can be used to add constraints between registers. Two registers can not be on the same cluster if they belong to a critical cycle with different skew. This has been explored during the thesis but it did not give better results. It needs a second chance.

It has been noticed that exists a critical core of registers. Other algorithms based on linear programming or SAT can be designed to solve the critical core of the problem, and assign the non critical ones with other heuristics.

Bibliography

[1] A. Davis and S.M. Nowick. An introduction to asynchronous circuit design. *The Encyclopedia of Computer Science and Technology*, 38.

[2] AL Davis. The architecture and system method of DDM1: A recursively structured Data Driven Machine. *Proceedings of the 5th annual symposium on Computer architecture*, pages 210–215, 1978.

[3] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, and Christos Sotiriou. Desynchronization: Synthesis of asynchronous circuits from synchronous specifications. *IEEE Transactions on Computer-Aided Design*, 25(10):1904–1921, October 2006.

[4] C.E. Leiserson and J.B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[5] J.P. Fishburn. Clock Skew Optimization. *IEEE Transactions on Computers*, 39(7):945–951, 1990.

[6] D.A. Joy and M.J. Ciesielski. Clock period minimization with wave pipelining. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 12(4):461–472, Apr 1993.

[7] KA Sakallah, TN Mudge, and OA Olukotun. CheckT c and minT c: timing verification andoptimal clocking of synchronous digital circuits. *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 552–555, 1990.

[8] N. Shenoy, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Graph algorithms for clock schedule optimization. *Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, pages 132–136, 1992.

[9] TG Szymanski. Computing optimal clock schedules. *Proceedings of the 29th ACM/IEEE conference on Design automation*, pages 399–404, 1992.

[10] T.G. Szymanski and N. Shenoy. Verifying clock schedules. *Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, pages 124–131, 1992.

[11] RB Deokar and SS Sapatnekar. A graph-theoretic approach to clock skew optimization. *Circuits and Systems, 1994. ISCAS'94., 1994 IEEE International Symposium on*, 1.

[12] Rahul B. Deokar and Sachin S. Sapatnekar. A fresh look at retiming via clock skew optimization. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 310–315, New York, NY, USA, 1995. ACM.

[13] Xun Liu, Marios C. Papaefthymiou, and Eby G. Friedman. Maximizing performance by retiming and clock skew scheduling. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 231–236, New York, NY, USA, 1999. ACM.

[14] K. Ravindran, A. Kuehlmann, and E. Sentovich. Multi-domain clock skew scheduling. *Computer Aided Design, 2003. ICCAD-2003. International Conference on*, pages 801–808, 2003.

[15] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–13, Washington, DC, USA, 1998. IEEE Computer Society.

[16] J.B. MacQueen. SOME METHODS FOR CLASSIFICATION AND ANALYSIS OF MULTIVARIATE OBSERVATIONS. 1966.

[17] J.A. Hartigan. *Clustering Algorithms*. John Wiley & Sons, Inc. New York, NY, USA, 1975.

[18] JA Hartigan and MA Wong. A K-means clustering algorithm. *JR Stat. Soc., Ser. C*, 28:100–108, 1979.

[19] MR Garey and D.S. Johnson. The Rectilinear Steiner Tree Problem in NP Complete. *SIAM Journal of Applied Mathematics*, 32(4):826–834, 1977.

[20] R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[21] J.B. Kruskal Jr. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[22] R.M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3):309–311, 1978.

[23] E.L. Lawler. *Combinatorial optimization*. Holt, Rinehart and Winston New York, 1976.

[24] S.M. Burns. *Performance analysis and optimization of asynchronous circuits*. PhD thesis, California Institute of Technology, 1991.

[25] J. Cochet-Terrasson, G. Cohen, and S. Gaubert. Numerical computation of spectral elements in max-plus algebra. *IFAC Conference on System Structure and Control*, 1998.

[26] Ali Dasdan, Sandy Irani, and Rajesh K. Gupta. Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems. In *Design Automation Conference*, pages 37–42, 1999.

[27] N. E. Young, Robert E. Tarjan, and J. B. Orlin. Faster parametric shortest path and minimum-balance algorithms. *Networks*, 21:205–221, 1991.

[28] R. Bellman. ON A ROUTING PROBLEM. 1956.

[29] LR Ford Jr. NETWORK FLOW THEORY. 1956.

[30] E.F. Moore. The shortest path through a maze. *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292, 1959.

[31] R.E. Tarjan. Shortest paths. Technical report, AT&T Bell Laboratories, 1981.

[32] B.V. Cherkassky and A.V. Goldberg. Negative-cycle detection algorithms. *Mathematical Programming*, 85(2):277–311, 1999.