UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

NICOLAS SILVEIRA KAGAMI

Monotonic Buffer Insertion

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engeneering

Advisor: Prof. Dr. André Inácio Reis

Porto Alegre
November 2015

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Graduação: Prof. Sérgio Roberto Kieling Franco
Diretor do Instituto de Informática: Prof. Luis da Cunha Lamb
Coordenador do Curso de Engenharia de Computação: Prof. Raul Fernando Weber
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

"If you want to make an apple pie from scratch,
you must first create the universe."
— Carl Sagan

ACKNOWLEDGEMENT

I would like to thank my family for their love and support throughout my life.

I owe Marina Rey and her family more than thanks, for lovingly welcoming me into their lives.

I would like to thank my advisor André Inácio Reis for sharing a bit of his wisdom with me. A special thanks to Jody Matos, who had great patience with me, and was always there when I needed.

I would like to thank Valter Roesler for granting me a great opportunity to work. I have always admired your integrity.

I must thank my colleague and friend William Lautenschläger, for always being available to bounce a few ideas off, despite the intense workload he has.

ABSTRACT

This document presents a multi-objective approach to buffer insertion. Our concept is applied to simple-cells-based circuits, extracted from AIGs. Node count minimization in AIGs tends to increase the logic sharing, which may imply in some fanout violations. The subsequent fanout limiting step can be used to determine more than just a fanout abiding cell, if few physical aspects are taken into consideration. The proposed method simultaneously provides preferential treatment to global critical paths and builds a placement-aware buffer structure.

Keywords: Buffer Insertion. Logic Synthesis.

Inserção de Buffer Monotônica

RESUMO

Este documento apresenta um conjunto de algoritmos que formam uma abordagem multi-objetivo para inserção de buffers. O conceito é aplicado a circuitos baseados em células simples, obtidos a partir de AIGs. A minimização de nodos em AIGs costuma aumentar o compartilhamento lógico, que pode implicar em violações de fanout. O passo de limitação de fanout subsequente pode definir mais do que uma célula com fanout restrito, se alguns aspectos físicos são levados em consideração. O método proposto limita o fanout da célula enquanto provendo tratamento especial para caminhos críticos globais e construindo uma árvore de inversores para limitação de fanout com conexões baseadas em posição.

Palavras-chave: Síntese Lógica,Inserção de Buffer,Limitação de Fanout.

LIST OF ABBREVIATIONS AND ACRONYMS

AIG	And Inverter Graph
PAIG	Positioned And Inverter Graph
DAG	Directed Acyclic Graph
EDA	Electronic Design Automation
VLSI	Very Large Scale Integration
AAG	ASCII AIG
PAAG	Positioned ASCII AIG
SDC	Synonpsys Design Constraints
LIB	Liberty
DEF	Design Exchange Format

LIST OF FIGURES

Figure 3.1 Pre-Processing: How the pre-delay (blue) and post-delay (green) are propagated from the I/O associated delays (a) throughout the circuit ($b \rightarrow c$). 19
Figure 3.2 Initial tree form, or "inverter spine". 23
Figure 3.3 Highest Delay Percentage: Disparate target delays produces a decent result. 24
Figure 3.4 Highest Delay Percentage: Similar target delays produces a bad result. 25
Figure 3.5 Average Delay Percentage: Disparate target delays produces a decent result. 26
Figure 3.6 Average Delay Percentage: Similar target delays produces a decent result, much better than Highest Delay Percentage results. 26
Figure 3.7 Inverter Group Difference: Vacant slots in the first level. 27
Figure 3.8 Inverter Group Difference: Trying to find the best group of four or less targets to be deemed critical. Y axis: Post-delay. X axis: Targets. 28
Figure 3.9 Inverter Group Difference: Two critical targets being allocated at the first layer and expanding the rest of the slots as inverters. 29

LIST OF TABLES

Table 4.1 Node order average delay values........36
Table 4.2 Critical allocation algorithm delay values........37
Table 4.3 Summed Inverter distances........38
Table 4.4 Summed Worst Inverter Tree distances........39

CONTENTS

1 INTRODUCTION......11
2 CONCEPTS......13
2.1 AIG......13
2.2 Manhattan Distance......13
2.3 Timing Budget......13
2.4 Consumers......14
2.5 Inverter Tree......14
3 METHODOLOGY......15
3.1 Inputs......16
3.1.1 Cell Library: Liberty......16
3.1.2 Topology: Placed ASCII AIG and Design Exchange Format......16
3.1.3 Timing Constraints: Synopsys Design Constraints......17
3.2 Pre-Processing......18
3.3 Order of operation......19
3.4 Tree Formation......22
3.5 Critical Selection and Insertion......23
3.5.1 Highest Delay Percentage Algorithm......23
3.5.2 Average Delay Percentage Algorithm......25
3.5.3 Inverter Group Difference Algorithm......27
3.6 Tree Expansion and Pruning......29
3.7 Non Critical Target Allocation......30
3.8 Tree Connection......31
3.8.1 K-Means Clustering Inspired Heuristic......31
3.8.2 Furthest First Heuristic......32
3.9 Inverter Positioning......34
3.10 Post-Processing......34
4 RESULTS......35
4.1 Testing Methodology......35
4.2 Delay Experiments......35
4.2.1 Node Order......35
4.2.2 Critical Algorithm......36
4.3 Distance Experiments......37
4.3.1 Tree Connection......38
5 CONCLUSION......40
REFERENCES......41

1 INTRODUCTION

The improvement in the semiconductor industry has been responsible for driving the quality and ubiquity of technology, which was partly brought forth by the scaling of transistor integration, also known as Moore's law (MOORE, 1965). This progress comes at the price of an increase in design complexity.

Modern Very-Large Scale Integration (VLSI) circuit design deals with this increase in complexity by relying on Electronic Design Automation (EDA) tools to optimally guide the design through the ever-increasing possibility space. One way to reduce this search space is to decrease the degrees of freedom, imposing design restrictions. Such is the way of semicustom design. A common semicustom approach is the cell-based design in which a standard cell library determines the building blocks upon which the logic structure is mapped.

The industry's evolving on cell-based designs has highlighted the important role of logic synthesis in the VLSI design flow. The logic synthesis step is the one responsible for the most powerful transformations on the logic implementing a given design (WAGNER; REIS; RIBAS, 2006). The literature is used to split logic synthesis into two major substeps: (1) technology-independent optimizations, in which the synthesis tasks are performed restrictively in the Boolean domain; and (2) technology-dependent optimizations, in which detailed information on the technology used to implement the logic elements are also taken into account (e.g. area, power and delay estimations obtained from the standard cell library). Examples of technology-independent optimizations can be found in the works by Callegaro et al. (2015), Matos and Reis (2015), Possani et al. (2015) and Neutzling et al. (2015). Concerning works on technology-dependent optimizations, please refer to the works by Alegretti et al. (2013), Machado et al. (2012) and Machado et al. (2013).

Regarding standard cell libraries, it is highly debated between defenders of small cell libraries composed of simple cells and defenders of larger cell libraries composed of more complex cells. Considering simple-cells based design, Matos et al. (2013) has shown an interesting approach to reducing the number of transistors by minimizing the number of nodes in an And-Inverter Graph (AIG) representation (MATOS et al., 2013). One of the effects of building a circuit from simple cells is an increase in logic sharing, and thus in fanout. This requires attention since a cell with fanout violations can compromise the circuit performance. This is solved by Matos et al. (2013) by means of a buffer

insertion algorithm which introduces an inverter tree to limit the fanout. For a more detailed explation on Matos el al. work, we refer the reader to Matos et al. (2013) and Matos (2014).

One of the complicating factors in the development of VLSI circuits is the growing importance of physical properties in the design. This is evidenced by the fact that, in older technologies, a proper timing estimation could be achieved solely from the logic gates, and thus, during a physically naive logic synthesis. This assessment by itself no longer results in a suitable inference of delay, which is essential for guiding the possible transformations in these earlier steps. According to Bakoglu (1990), when all dimensions of a device are scaled down by a factor of S, the intrinsic delay is lowered by a factor of S, while the local interconnect delay remains the same and the global interconnect delay increases by a factor of S^2. Along the industry's development, this consequence of shrinking dimensions reassigned the prevalence of intrinsic delay to interconnect delay. There are several adjustments that can take place during logic synthesis which inevitably influence some of the subsequent physical synthesis properties (MATOS; REIS, 2015; MATOS et al., 2015). Consequently, an argument could be made for increasing physical awareness during logic synthesis, especially wiring awareness.

This work presents a buffer insertion algorithm based on Matos et al. (2013), aiming to consider its connective structure with a greater rigor, plainly establishing the wiring connections necessary for its inverter tree. Such is made possible by considering the positions of nodes in a placed AIG configuration. The algorithm uses this very representation of delays together with an associated I/O timing budget to determine the inverter tree structure such that critically delayed targets receive preferential treatment.

The ensuing chapters are organized as follows. Chapter 2 defines some of the significant concepts for understanding the buffer insertion algorithm. Chapter 3 displays the inner workings of the buffer insertion algorithm. Chapter 4 shows our testing methodology as well as the results obtained. Chapter 5 outlines our contribution.

2 CONCEPTS

This chapter introduces some of the fundamental concepts that will be regularly used in the following chapters.

2.1 AIG

An And-Inverter Graph is a Directed Acyclic Graph (DAG) that represents a structural instance of a logical function. AIGs are composed of AND2 nodes, possibly negated edges and terminal variable nodes. This concept has been found to be an efficient approach at functional representation for EDA applications (DARRINGER et al., 1981). AIGs are highly used data structure in logic synthesis, especially during the early steps of the design flow. At this phase, area is minimized by decreasing the total number of nodes in the AIG, while delay is minimized by reducing the depth of the AIG graph.

Placed AIGs are AIGs where each node has an assigned (X, Y) position. Placed AIGs allow us to consider physical layout information during the early steps of logic synthesis (MATOS; REIS, 2015).

2.2 Manhattan Distance

The Manhattan Distance between two points is defined as the sum of the absolute differences of their Cartesian coordinates (KRAUSE, 1987). This concept is useful for VLSI routing, as it properly characterizes the distance between points when connecting through horizontal and vertical segments. Keep in mind that whenever this work uses spatial terms like "distance", "close", "closest, "far", "furthest", we will be referring to Manhattan Distance.

2.3 Timing Budget

A timing budget is the assignment of arrival times for each I/O pin of entities instanced in a network. This is a crucial parameter for the proposed method of buffer insertion, as each node abstraction in the circuit keeps an associated timing budget. This budget is composed of two values explained below.

- The *Pre-Delay* is an estimate of the worst signal arrival time in this node from the primary inputs[1].
- The *Post-Delay* is an estimate of the worst signal propagation time from the current node to the primary outputs[1].

2.4 Consumers

In the context of this work, a node's consumer is an entity which requires to be fed that node's signal. A consumer can be positive or negative, depending on its requirements, which are defined by circuit topology. A positive consumer requires the node's signal while a negative consumer requires its negated signal.

2.5 Inverter Tree

An inverter tree is the structure used in the proposed approach to limit the fanout of the cells whilst supplying all of its consumers. An inverter tree is composed of inverters which feed into other inverters or consumers. The degree of this tree is defined as the inverter maximum fanout. As long as the degree is greater than one, the number of vacant signals produced by the inverters can increase with each level, eventually allowing a tree capable of feeding any given number of consumers.

[1]Since this work is focused on combinational circuits, no sequential elements are considered in this analysis.

3 METHODOLOGY

In this chapter, we present the inner workings of the Buffer Insertion algorithm developed. The main objective of such process is to compose an inverter tree capable of supplying all node's positive and negative consumers without violating a given fanout limit. This imposes a decision about how to construct this inverter tree. In an effort to minimize the critical path delay of the circuit, we will apply heuristics to determine the connections and positions of the inverters and how they supply the targets.

The targets are sorted by post-delay and divided into critical and non-critical targets. Critical targets with higher delay will be allotted preferably closer to the root, in detriment to the delay of the non-critical targets. The tree is then expanded to be able to supply all targets and the non-critical targets are distributed along the leaves. The targets are appointed to inverters according to a locality heuristic.

,

Algorithm 3.1: Monotonic Buffer Insertion

```
1 MonotonicBufferInsertion(AIG_File,DEF_File,LIB_File,SDC_File){
2     Parse(AIG_File,DEF_File,LIB_File,SDC_File);
3     PreProcessing();
4     NodeProcessing();
5     PostProcessing();
```

Algorithm 3.2: Inserting buffer for a node

```
1 InsertBuffer(node){
2     FormTree(node);
3     CriticalAllocation(node);
4     Expand(node);
5     Prune(node);
6     NonCriticalAllocation(node);
7     ConnectTree(node);
```

Section 3.1 presents the information necessary to run the algorithm and how this information is obtained. Section 3.2 details the process responsible for seeding the initial delay values throughout the graph. Section 3.3 proposes some implications of the order of execution. Section 3.4 describes the initial inverter tree abstraction. Section 3.5 displays the algorithms employed to select and allocate critical targets. Section 3.6 delineates the transformations the inverter tree goes through between the critical allocation and non-critical allocation steps. Section 3.7 defines the method responsible for non-critical target

allocation. Section 3.8 demonstrates the heuristics used for defining the inverter tree connections. Section 3.9 discloses the inverter positioning algorithms. Section 3.10 depicts the process responsible for determining the final delay values throughout the graph.

3.1 Inputs

The Monotonic Buffer Insertion requires a positioned consumer graph, an I/O timing budget and the delay and maximum fanout of the nodes. The graph is determined by the circuit topology, to be extracted either from a PAAG or a DEF file. The I/O associated timing budget is defined by an SDC file, which also defines the target periodicity of the design. A cell library is parsed in order to determine several pieces of information about the cells, mainly the delay.

Algorithm 3.3: Parsing

```
1 Parse(AIG_File, DEF_File, LIB_File, SDC_File){
2     if (AIG_File) then
3         Topology = TopologyFromAIG(AIG_File);
4     else
5         Topology = TopologyFromDEF(DEF_File);
6     CellLibrary = ParseLIB(LIB_File);
7     TimingConstraints = ParseSDC(SDC_File);
```

3.1.1 Cell Library: Liberty

The LIB file defines a group of cells in terms of its pins, drive strength, area, estimated delay and power consumption. This information is appropriate for establishing several pertinent information about the circuit after the application of the buffer insertion.

3.1.2 Topology: Placed ASCII AIG and Design Exchange Format

The PAAG file defines an And-Inverter Graph (AIG) whose nodes are placed. The AIG directly determines the node's positive and negative consumers. The placed graph is trivially extractable from AIGs by applying on it any state-of-the-art placement algorithm. Concerning the DEF file, we are considering only a small set of DEF primitives

used to define placed components (instances of cells), pins (output and input) and nets (connections). This information needs to be processed in order to extract the logical arrangement consisting of each node's positive and negative consumers.

3.1.3 Timing Constraints: Synopsys Design Constraints

The SDC file considered in this work defines the target clock cycle, the input pins' arrival time relative to the clock, and the maximum delay restriction for each output, considering different timing paths. This is crucial information for defining the timing budget, and therefore can heavily influence the overall structure resulted from the buffer insertion.

3.2 Pre-Processing

The pre-processing stage takes the input and output timing information acquired from the file parsers and propagates it throughout the circuit. This step establishes the initial estimates for the pre-delay and post-delay values for each node in the graph. These values are assessed cumulatively according to the nodal delay and the outer timing constraints.

Algorithm 3.4: Pre-Processing

```
1  PreProcessing(){
2      foreach (node in Topology) do
3          node.target.pathDelay = 0;
4      EstimateDelay();
```

Algorithm 3.5: Estimating delay for the network

```
1  EstimateDelay(){
2      NodeQueue = new Queue;
3      NodeQueue.empty();
4      foreach (node in Topology.inputs) do
5          NodeQueue.push(node);
6      while (!NodeQueue.empty()) do
7          foreach (target in node.targets) do
8              preDelayEstimate = computePreDelay(node,target);
9              if (preDelayEstimate > target.preDelay) then
10                 target.preDelay = preDelayEstimate;
11                 NodeQueue.push(target);
12     NodeQueue.empty();
13     foreach (node in Topology.outputs) do
14         NodeQueue.push(node);
15     while (!NodeQueue.empty()) do
16         foreach (source in node.sources) do
17             postDelayEstimate = computePostDelay(node,source);
18             if (postDelayEstimate > source.postDelay) then
19                 source.postDelay = postDelayEstimate;
20                 NodeQueue.push(source);
```

Figure 3.1 shows how the initially I/O associated delays are propagated throughout the circuit, taking the nodal delays into consideration.

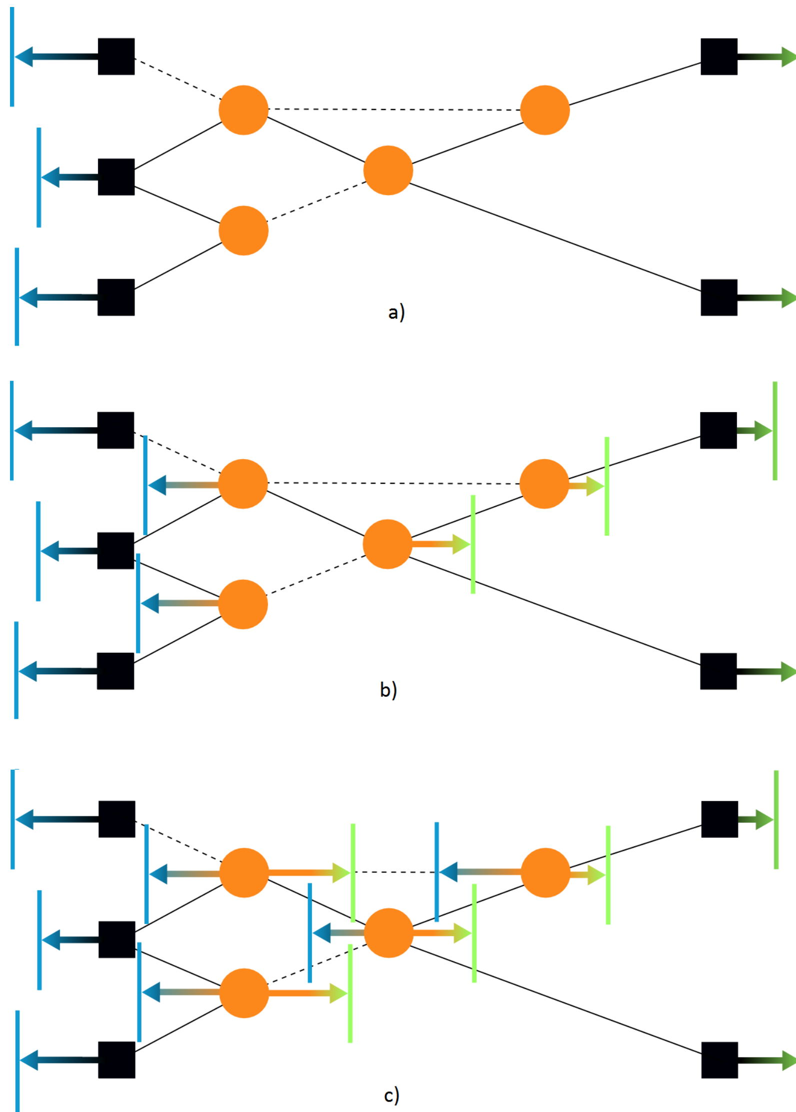

Figure 3.1 – Pre-Processing: How the pre-delay (blue) and post-delay (green) are propagated from the I/O associated delays (a) throughout the circuit ($b \rightarrow c$).
Source: The author

3.3 Order of operation

The order in which the nodes are processed is relevant because it affects the information available at each processing step. Every time a node goes through the buffer insertion process we can have a better estimate of the path delay to its targets. This information can be used to more adequately measure the post-delay of the processed node as well as the pre-delays of its targets. Thus, after a node has been processed, its post-delay is updated in accordance with the delay introduced by the inverter tree. Since the algorithm takes only the position and post-delay of its consumers into consideration, we don't yet update the pre-delay of its targets. The updated post-delay values allow for a more in-

formed buffer insertion for neighbor nodes that use this data. Therefore the order in which we apply the buffer insertion over the nodes can have an impact on the circuit's maximum delay, lest we repeat nodes. Three node navigation approaches have been developed for this stage:

- A *Simple Iteration* over all the nodes in the order they are stored in memory, indifferent to which of it's neighbors have been processed. This method is very simple and shows good results.
- The *Input Propagation* is a queue which starts at the inputs and only processes nodes whose sources have been processed.
- The *Output Propagation* is a queue which starts at the outputs and only processes nodes whose targets have been processed.

Algorithm 3.6: Node Processing

```
1 NodeProcessing(){
2     switch (ProcessingOrder)do
3         case (Iteration)
4             IterateNodes();
5         case (InputPropagation)
6             InputPropNodes();
7         case (OutputPropagation)
8             OutputPropNodes();
```

Algorithm 3.7: Iteration node processing

```
1 IterateNodes(){
2     foreach (node in Topology)do
3         InsertBuffer(node);
```

Considering we take the post-delay into account to construct the inverter tree, the propagation from the outputs should have an advantage because every node will always have its targets' post delay determined. By the same logic the queue which starts at the inputs will lose that advantage. The input propagation option was developed not only to show this effect, but also for a possible future enhancement where the buffer insertion takes its target's pre-delay into account.

Algorithm 3.8: Input Propagation node processing

```
1  InputPropNodes(){
2      NodeQueue = new Queue;
3      NodeQueue.empty();
4      foreach (node in Topology) do
5          Expanded[node] = false;
6      foreach (node in Topology.inputs) do
7          NodeQueue.push(node);
8      while (!NodeQueue.empty()) do
9          node = NodeQueue.pop();
10         InsertBuffer(node);
11         Expanded[node] = true;
12         foreach (target in node.targets) do
13             Ready = true;
14             foreach (source in target.sources) do
15                 if (Expanded[source] == false) then
16                     Ready = false;
17             if (Ready == true) then
18                 NodeQueue.push(target);
```

Algorithm 3.9: Output Propagation node processing

```
1  OutputPropNodes(){
2      NodeQueue = new Queue;
3      NodeQueue.empty();
4      foreach (node in Topology) do
5          Expanded[node] = false;
6      foreach (node in Topology.outputs) do
7          NodeQueue.push(node);
8      while (!NodeQueue.empty()) do
9          node = NodeQueue.pop();
10         InsertBuffer(node);
11         Expanded[node] = true;
12         foreach (source in node.sources) do
13             Ready = true;
14             foreach (target in source.targets) do
15                 if (Expanded[target] == false) then
16                     Ready = false;
17             if (Ready == true) then
18                 NodeQueue.push(source);
```

3.4 Tree Formation

The first step in the buffer insertion is the formation of the tree. The initial tree representation considers only the circumstances of each level's slots. These slots represent places where an inverter or a target can be supplied. Therefore, each level maintains the amount of vacant slots, the amount taken by inverters and the amount taken by targets. The rest of the inverter tree information, such as the position of each inverter and the targets and/or inverters it supplies, will be defined later in the process. The tree starts at the minimum height required by the number of positive and negative consumers.

Algorithm 3.10: Tree Formation

```
1  FormTree(node){
2      minHeight = computeMinimumHeight(node);
3      tree = new inverterTree(minHeight);
4      tree.levels[0].vacant = CellMaxFanout -1;
5      tree.levels[0].inverterTaken = 1;
6      for (h=1;h<minHeight;h++)do
7          tree.levels[h].vacant = InverterMaxFanout -1;
8          tree.levels[h].inverterTaken = 1;
9          tree.levels[h].signalTaken = 0;
10     node.inverterTree = tree;
```

Notice that this will likely not be the actual height of the final inverter tree since the critical allocation algorithms will usually insert the priority targets as close to the root as possible. This will effectively increase the height of the tree and the number of inverters in the path to the non-critical targets. By the very fact the critical algorithms will begin delegating targets to the levels closer to the root, we must reserve slots to ensure the tree's capability of feeding all of its targets. This is accomplished by having each level start with feeding an inverter to the next level. The result is a so called "inverter spine" as can be seen in figure 3.2. After this process, the tree is ready to receive its targets.

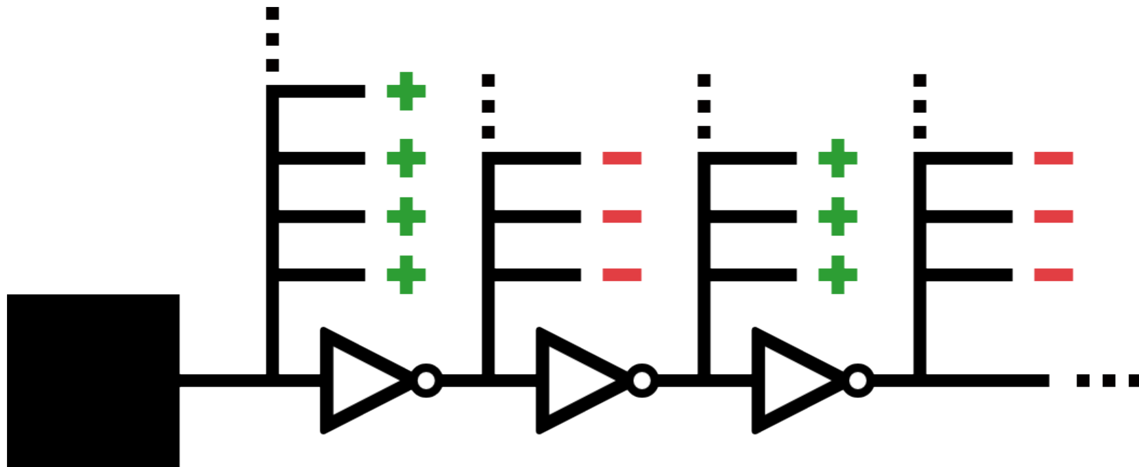

Figure 3.2 – Initial tree form, or "inverter spine".
Source: The author

3.5 Critical Selection and Insertion

The critical selection determines which of the consumers deserve special treatment. Three techniques were developed to both assess the criticality and designate the level the targets occupy in the inverter tree. All of them depend on the post-delay of the target. For this reason, the targets are arranged in the order of highest to lowest before running these heuristics.

Algorithm 3.11: Critical Selection and Insertion

```
1 CriticalAllocation(node){
2     node.targets.sort();
3     switch (CriticalAllocationAlgorithm)do
4         case (HighestDelayPercentage)
5             HighestDelayAlgorithm();
6         case (AverageDelayPercentage)
7             AverageDelayAlgorithm();
8         case (InverterGroupRelative)
9             InverterGroupAlgorithm();
```

3.5.1 Highest Delay Percentage Algorithm

This algorithm deems a target critical if its post-delay is within a percentage of the highest target delay. Consequently, this technique will always generate at least one

critical. The critical targets are then individually assigned in order of criticality from the root up. This is a fast and simple method which has weaknesses. If all targets are close in terms of post-delay they will all be considered critical and this approach will generate a considerably worse scenario than if none were critical. Figure 3.3 shows an example of Highest Delay Percentage critical selection when the targets have disparate post-delay values. Figure 3.4 shows an example of critical selection when the targets have similar post-delay values. Notice that multiple targets have been selected as critical.

Algorithm 3.12: Highest Delay Percentage Critical algorithm

```
1 HighestDelayAlgorithm(node){
2     highestDelay = node.targets[0].postDelay;
3     foreach (target in node.targets)do
4         if (target.postDelay >= highestDelay*delayPercentage)then
5             node.inverterTree.insertCritical(target);
```

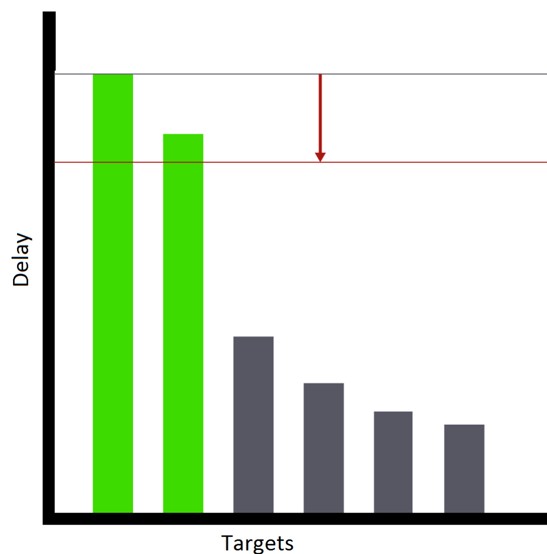

Figure 3.3 – Highest Delay Percentage: Disparate target delays produces a decent result.
Source: The author

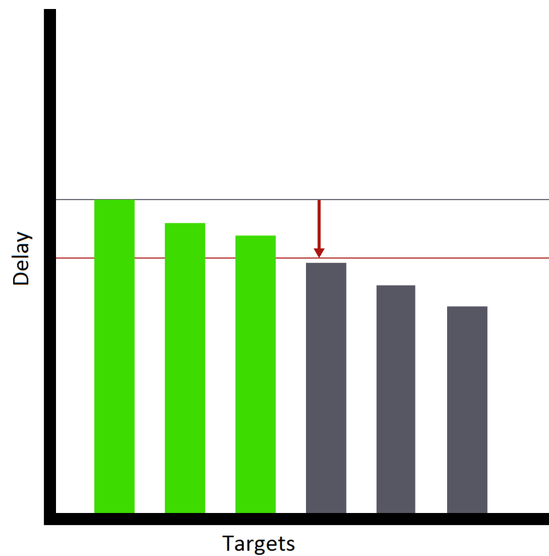

Figure 3.4 – Highest Delay Percentage: Similar target delays produces a bad result.
Source: The author

3.5.2 Average Delay Percentage Algorithm

This algorithm deems a target critical if its post-delay is more than a certain factor above the average post-delay of the node's targets. The critical targets are then individually assigned in order of criticality from the root up.

Algorithm 3.13: Average Delay Percentage Critical algorithm

```
1 AverageDelayAlgorithm(node){
2     averageDelay = 0;
3     foreach (target in node.targets)do
4         averageDelay += targets.postDelay;
5     averageDelay /= node.numTargets;
6     foreach (target in node.targets)do
7         if (target.postDelay >=
          averageDelay*(1+delayPercentage))then
8             node.inverterTree.insertCritical(target);
```

This method is almost as fast as the last one, but is immune to its most critical flaw. A group of similarly delayed targets will, by definition, not be a considerable factor above the average. This factor can be defined relative to the proportion of the inverter delay to the cell delay. Such that a target deserves to be treated specially if its delay exceeds this factor above average. This algorithm could possibly generate some negative results if some of the node's targets have very small post-delays, bringing the average post-delay

down and hence assuming a few more criticals than ideal. 3.5 shows an example of Average Delay Percentage critical selection when the targets have disparate post-delay values, while figure 3.6 shows an example of when the targets have similar post-delay values. Notice this latter result is quite different when compared to the Highest Delay Percentage.

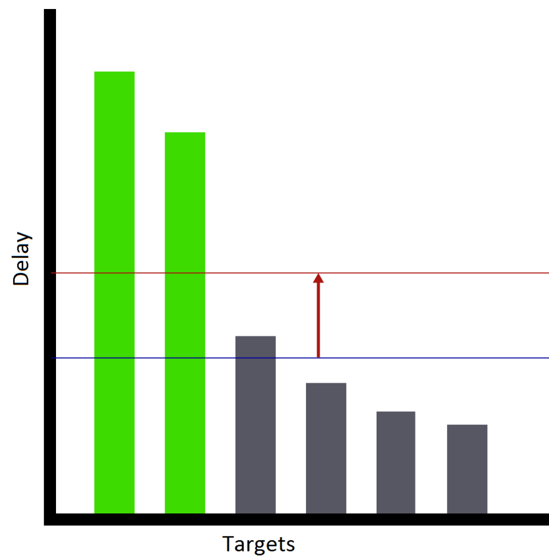

Figure 3.5 – Average Delay Percentage: Disparate target delays produces a decent result.
Source: The author

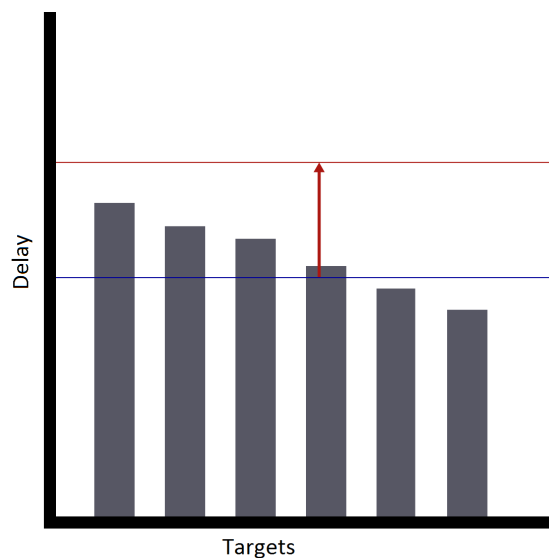

Figure 3.6 – Average Delay Percentage: Similar target delays produces a decent result, much better than Highest Delay Percentage results.
Source: The author

3.5.3 Inverter Group Difference Algorithm

This algorithm relies in the difference between the targets' post-delay measured in inverter delay units. In theory, a target "deserves" to be placed a layer above other targets if its difference is greater than the delay of the layer. This difference is two times the inverter delay, as we must match the polarity of the consumer with its assigned layer, and so, the "next available layer" is always two inverters away.

Algorithm 3.14: Inverter Group Relative Critical algorithm

```
1  InverterGroupAlgorithm(node){
2      averageDelay = 0;
3      tree = node.inverterTree;
4      foreach (target in node.targets)do
5          averageDelay += targets.postDelay;
6      foreach (level in tree.levels)do
7          groupSize = level.vacant;
8          for
           (n=tree.numCritical+groupSize;n>tree.numCritical;n − −)do
9              if ((tree.targets[n].postDelay - tree.targets[n+1].postDelay)
               > 2*InverterDelay)then
10                 for (c=tree.numCritical;c<n;c++)do
11                     tree.insertCritical(tree.targets[c]);
```

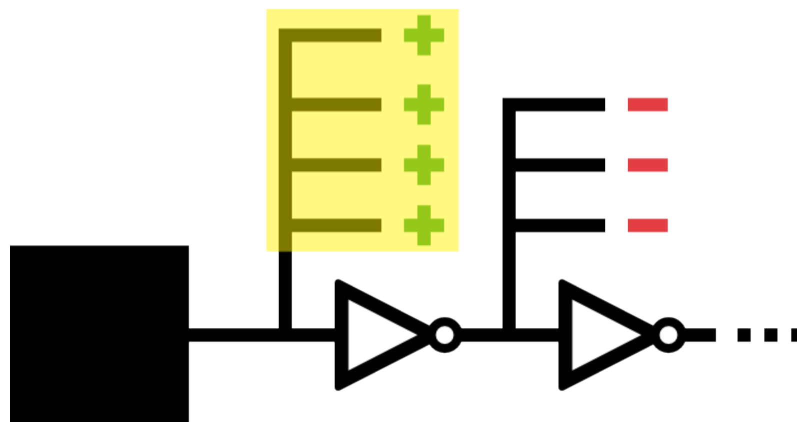

Figure 3.7 – Inverter Group Difference: Vacant slots in the first level.
Source: The author

This algorithm iterates over the levels of the inverter tree, starting at the root. At each level the algorithm calculates the number of vacant slots v and tries to find the biggest group with size smaller or equal to v such that every target belonging to this group has a post-delay greater than this standard distance. The remaining slots are filled with inverters, generating more vacant slots in the next level.

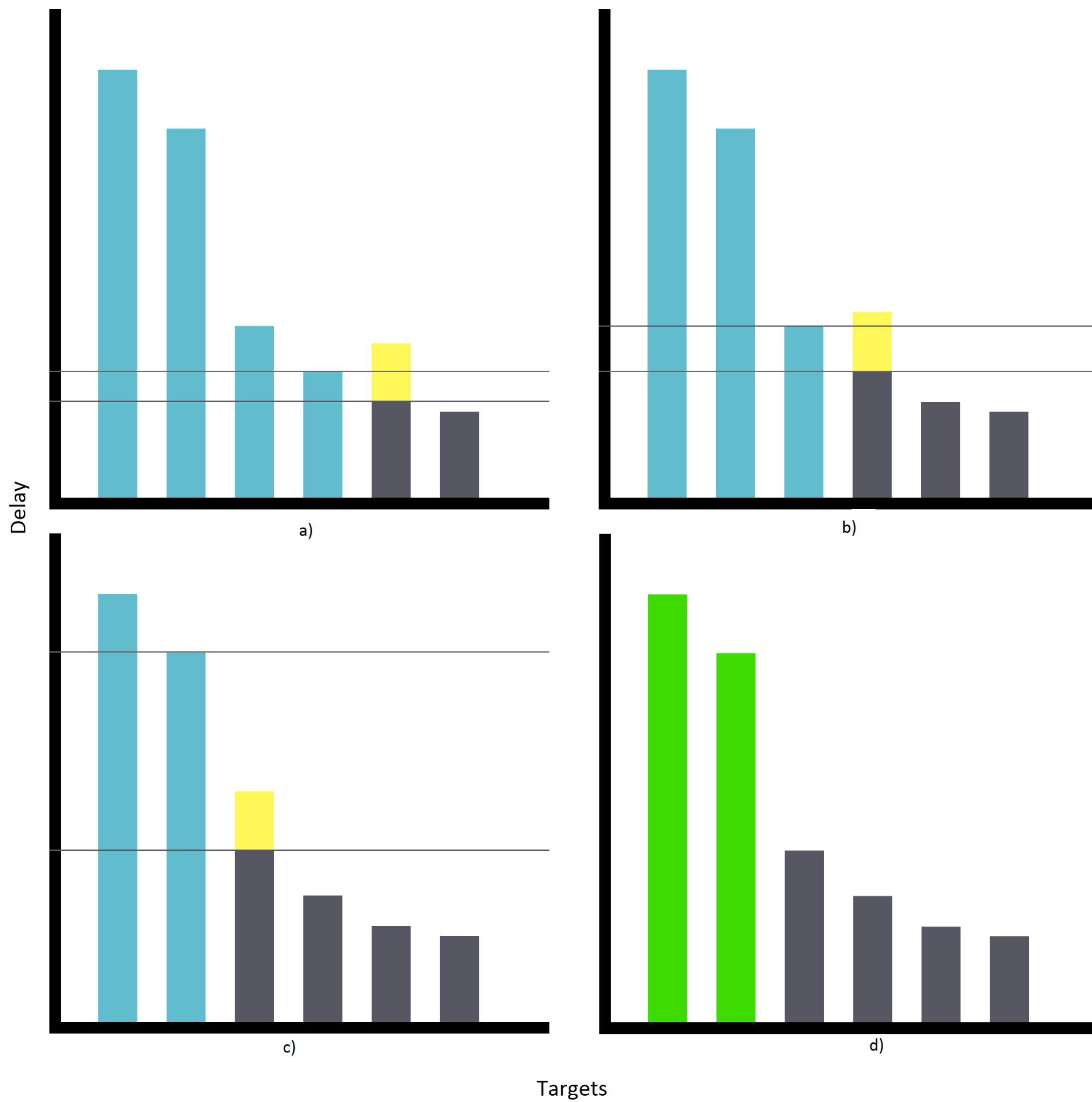

Figure 3.8 – Inverter Group Difference: Trying to find the best group of four or less targets to be deemed critical. Y axis: Post-delay. X axis: Targets.

Source: The author

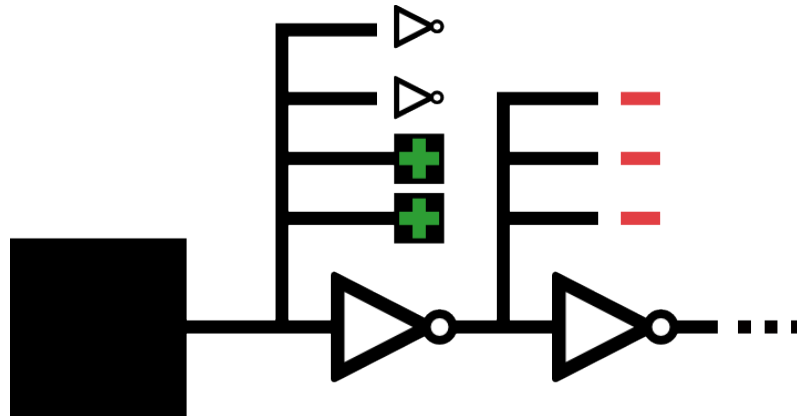

Figure 3.9 – Inverter Group Difference: Two critical targets being allocated at the first layer and expanding the rest of the slots as inverters.
Source: The author

Figure 3.7 depicts four vacant slots in the first layer. Figure 3.8 shows the algorithm determining the best group of critical targets. Figure 3.9 displays the result of this selection, two critical targets and two inverters to feed more signals in the next layer.

3.6 Tree Expansion and Pruning

After the critical targets have been assigned to specific levels the tree is left at a state where it will probably need more leaves to supply the non-critical targets. Thus, the tree goes through a process where inverters are added from the root wherever there are vacant slots. This growth is maintained until it reaches a level whereby all remaining targets can be supplied. Even though the tree has reached its now requisite height, not all non-critical targets need to be arranged at the last two levels. A pruning algorithm follows the expansion to get rid of the excess inverters brought about by the expansion.

Algorithm 3.15: Tree Expansion

```
1 Expand(){
2     while (positiveConsumersLeft<positiveSlotsAvailable &&
      negativeConsumersLeft<negativeSlotsAvailable)do
3         tree.addLevel();
4         calculateBestCaseAvailableSlots();
```

Algorithm 3.16: Tree Pruning

```
1  Prune(){
2      do
3          invSaved = 0; for (l=tree.numLevels;l>0;l − −)do
4              invSaved += tree.level[1].removeExcess();
5          rearrangeTargets();
6      while (invSaved > 0);
```

3.7 Non Critical Target Allocation

The non critical targets get distributed along the available slots on the tree, from the root up, in the order of highest to lowest delay. And now we have a level assigned to each of the targets. If this step didn't take the targets' delays into consideration it would be equivalent to a less informed buffer insertion.

Algorithm 3.17: Non-Critical Selection and Insertion

```
1  NonCriticalAllocation(){
2      for (h=0;h<minHeight;h++)do
3          if ((h%2)==0)then
4              for (t=0;t<node.inverterTree.levels[h].vacant;t++)do
5                  node.inverterTree.insertNextPositiveTarget(h);
6              node.inverterTree.levels[h].vacant = 0;
7              node.inverterTree.levels[h].signalTaken += t;
8          else
9              for (t=0;t<node.inverterTree.levels[h].vacant;t++)do
10                 node.inverterTree.insertNextNegativeTarget(h);
11             node.inverterTree.levels[h].vacant = 0;
12             node.inverterTree.levels[h].signalTaken += t;
```

3.8 Tree Connection

This stage defines the structure of the inverter tree to a greater detail. It is responsible for defining which targets will be fed by which inverters and which inverters will feed those inverters and so forth until the root. The main objective of this step is to quickly and roughly connect targets by locality such that similarly placed targets get fed by the same inverter, avoiding long connection delays. Two algorithms were developed to carry out this task.

Algorithm 3.18: Tree connection algorithms

```
1 ConnectTree(node){
2     switch (TreeConnectionAlgorithm) do
3         case (Lloyds)
4             LloydsConnection();
5         case (FurthestFirst)
6             FurthestFirst();
```

3.8.1 K-Means Clustering Inspired Heuristic

The K-means clustering is a problem of partitioning a number of points into k clusters. A commonly applied algorithm gets called K-means algorithm or Lloyd's algorithm. Our implementation was roughly based on this algorithm. These k points will represent the k inverters and the associated points represent the targets. It starts by positioning k points on top of random existing targets. Then every target is assigned to its closest k point respecting the inverter's maximum fanout. Then every inverter is repositioned according to its associated targets. The last two steps can be repeated until an arbitrary condition is met. For example, when there is a net decrease in the sum of distances between the targets and their appointed inverter drops below a certain threshold. A weakness this approach presents is that there is no guarantee that the furthest point from the source will manage to reach its best positioned inverter before its fanout is full, possibly aggravating an already onerous connection.

Algorithm 3.19: Lloyds tree connection algorithm

```
1  LloydsConnection(node){
2      foreach (level in node.inverterTree.levels)do
3          for (i=0;i<level.numInverters;i++)do
4              inverter[i] = new Inverter;
5              inverter[i].position = level.targets[random].position;
6          for (iteration=0;iteration<NumIterations;iteration++)do
7              for (i=0;i<level.numInverters;i++)do
8                  inverter[i].clearTargets();
9              foreach (target in level.targets)do
10                 inv = findClosestInverter(target.position);
11                 inv.addTarget(target);
12             for (i=0;i<level.numInverters;i++)do
13                 inverter[i].positionInverter();
```

3.8.2 Furthest First Heuristic

The furthest first inverter allocation is a quite simple heuristic that tries to ensure that the furthest target from the source gets a good chance at an inverter slot. It starts by determining the furthest target and positions an inverter there, connected to the target. This inverter then searches for its closest targets until it supplies its quota. We repeat these last two steps, disregarding the supplied targets.

Algorithm 3.20: Furthest First tree connection algorithm

```
1  FurthestFirst(node){
2    foreach (level in node.inverterTree.levels)do
3      for (i=0;i<level.numInverters;i++)do
4        foreach (target in level.targets)do
5          supplied[target] = false;
6        worstDistance = 0;
7        foreach (target in level.targets)do
8          if ((supplied[target] == false) && (worstDistance <
           distance(target.position,node.position)))then
9            worstDistance =
             distance(target.position,node.position);
10           furthestTarget = target;
11       inverter[i] = new Inverter;
12       inverter[i].position = furthestTarget.position;
13       inverter[i].addTarget(furthestTarget);
14       for (i=1;i<InverterMaxFanout;i++))do
15         closestDistance = worstDistance;
16         foreach (target in level.targets)do
17           if ((supplied[target] == false) && (closestDistance >
             distance(target.position,inverter.position)))then
18             closestDistance =
               distance(target.position,inverter.position);
19             closestTarget = target;
20         inverter[i].addTarget(closestTarget);
21       inverter[i].positionInverter();
```

3.9 Inverter Positioning

Both of the tree connection algorithms developed require a phase whereby an inverter must be positioned according to its target points. Two simple methods were designated to define the inverter position.

- The *Centroid* point, defines the position as the average of its targets coordinates.
- The *Delay Weighted Centroid* point, which defines the position as a weighted average of the target coordinates according to each targets' post-delay.

3.10 Post-Processing

The post processing stage aims to determine the pre-delay and post-delay values now that the delay from producer node to consumer node is more accurately represented. In a process much like the pre-processing, delays are propagated and recalculated in order to determine the critical path delay, which is defined as the highest individual sum of pre-delay and post-delay.

Algorithm 3.21: Post-Processing

```
1 PostProcessing(){
2     EstimateDelay();
3     worstDelay = 0;
4     foreach (node in Topology)do
5         summedDelay = node.preDelay + node.postDelay + NodalDelay;
6         if (summedDelay > worstDelay)then
7             worstDelay = summedDelay;
```

4 RESULTS

This chapter presents the results obtained when trying to determine the effectiveness of some of the algorithms developed, as well as the testing methodology. The methods studied in this chapter include the Critical Insertion algorithms, the Tree Connection algorithms and the Node Navigation algorithms.

4.1 Testing Methodology

The test set selected to apply the buffer insertion algorithms consists of ten combinational benchmark circuits which were rewritten after being transformed into PAIGs (BRGLEZ; FUJIWARA, 1985).

4.2 Delay Experiments

This experiment calculated the critical path delay, measured in delay units. Both the cells and the inverters contributed one unit to the delay. Multiple instances of the buffer insertion algorithms were applied to the test set, considering all possible combinations of the critical allocation algorithms as well as the processing order of nodes. The objective of this experiment is to compute the influence of applying these algorithms as opposed to treating all consumers indifferently.

4.2.1 Node Order

The table 4.1 shows the average critical delay resulted from applying different orders of node processing permutated over all critical allocation algorithms.The column entitled "Random" represents the values obtained without applying a critical allocation algorithm and randomly distributing the targets along the leaves of the inverter tree. The "Iteration" column shows the results of processing the nodes in the order they appear in the memory. The "Input Prop" column shows the result of starting at the nodes connected to inputs and only propagating to other nodes when its sources have been processed. The "Output Prop" column shows the result of starting at the nodes connected to output and only propagating to other nodes when its consumers have been processed. The "Gain"

columns show the percentage gains of the values in the respective left column in relation to the "Random" values.

Table 4.1 – Node order average delay values

Circuit	*Random*	*Iteration*	*Gain*	*Input Prop*	*Gain*	*Output Prop*	*Gain*
C432	61.00	59.67	2.19%	59.00	3.28%	59.67	2.19%
C499	57.00	55.67	2.34%	55.67	2.34%	55.67	2.34%
C880	79.00	60.33	23.63%	60.33	23.63%	60.33	23.63%
C1355	57.00	52.33	8.19%	54.33	4.68%	52.00	8.77%
C1908	77.00	73.00	5.19%	73.33	4.76%	73.00	5.19%
C2670	61.00	56.33	7.65%	56.00	8.20%	56.33	7.65%
C3540	104.00	88.00	15.38%	88.67	14.74%	87.67	15.71%
C5315	74.00	69.00	6.76%	69.33	6.31%	69.33	6.31%
C6288	223.00	225.00	-0.90%	228.33	-2.39%	222.33	0.30%
C7552	108.00	109.67	-1.54%	109.67	-1.54%	108.67	-0.62%
Average Gain:			6.89%		6.40%		7.15%

Source: The author

The results displayed in the table 4.1 show that, on average, performance was a little bit worse propagating from the inputs relative to simply iterating over the nodes. Propagation from the outputs was the best option. This makes sense, considering that the buffer insertion algorithm takes into consideration the estimated delay posterior to its consumers. This delay is updated everytime a node is processed. Consequently a node whose consumers have been processed operates over more accurate data. Propagating from the outputs maximizes this effect and propagating from the inputs minimizes it.

4.2.2 Critical Algorithm

The table 4.2 shows the average critical delay resulted from applying different critical allocation algorithms permutated over all net order algorithms. The column entitled "Random" represents the values obtained without applying a critical allocation algorithm and randomly distributing the targets along the leaves of the inverter tree. The "Highest" column shows the results of applying the Highest Delay Percentage algorithm. The "Average" column shows the result of applying the Average Delay Percentage algorithm. The "Inv Difference" column shows the result of applying the Inverter Group Difference algorithm. The "Gain" columns show the percentage gains of the values in the respective left column in relation to the "Random" values.

Table 4.2 – Critical allocation algorithm delay values

Circuit	*Random*	*Highest*	*Gain*	*Average*	*Gain*	*Inv Difference*	*Gain*
C432	61.00	62.00	-1.64%	59.33	2.73%	57.00	6.56%
C499	57.00	57.00	0.00%	54.00	5.26%	56.00	1.75%
C880	79.00	59.00	25.32%	59.00	25.32%	63.00	20.25%
C1355	57.00	53.33	6.43%	52.67	7.60%	52.67	7.60%
C1908	77.00	73.33	4.76%	73.00	5.19%	73.00	5.19%
C2670	61.00	55.67	8.74%	56.00	8.20%	57.00	6.56%
C3540	104.00	87.67	15.71%	87.00	16.35%	89.67	13.78%
C5315	74.00	68.00	8.11%	68.00	8.11%	71.67	3.15%
C6288	223.00	235.00	-5.38%	220.33	1.20%	220.33	1.20%
C7552	108.00	103.00	4.63%	122.33	-13.27%	102.67	4.94%
Average Gain:			6.67%		6.67%		7.10%

Source: The author

The results shown in 4.2 demonstrate that the Inverter Group Difference algorithm had the best results on average. In spite of both the Highest and Average algorithms having a better performance in some cases, there are instances where this approach resulted in a longer critical path than the one produced by randomly distributing the targets. The Inverter Group Difference algorithm was the only one to guarantee a better result than the random alternative. This is likely due to a more cautious approach taken by the Inv Difference algorithm. The other algorithms had clear vulnerabilities, perhaps more clearly observed in a few considerably negative contributions.

4.3 Distance Experiments

This experiment measured distances between the inverters and their targets. The objective of this experiment is to determine the efficiency of the tree connection algorithms. The "Random" column depicts the outcome of randomly delegating targets to inverters at each layer of the inverter tree. The "Kmeans" column shows the results of applying our K-means inspired algorithm. The "WorstFirst" column shows the results of applying the Worst First algorithm. The "Gains" columns show the percentage reduction of the value on the respective left column in relation to the "Random" column.

4.3.1 Tree Connection

The table 4.3 below presents the accumulated distances between all inverters and their targets. It paints a clear picture that the K-means is substantially better than the random alternative. The Worst First, on the other hand, performed considerably worse at the sum of all inverter differences.

Table 4.3 – Summed Inverter distances

Circuit	*Random*	*Kmeans*	*Gain*	*WorstFirst*	*Gain*
C432	442608	392941	11.22%	777280	-75.61%
C499	2392242	2234692	6.59%	3089320	-29.14%
C880	2361505	2046822	13.33%	2843920	-20.43%
C1355	2480707	2153005	13.21%	2911200	-17.35%
C1908	2016797	1947048	3.46%	2583280	-28.09%
C2670	3456561	2876101	16.79%	4474310	-29.44%
C3540	9955270	8780764	11.80%	14504920	-45.70%
C5315	12274852	9888368	19.44%	14611150	-19.03%
C6288	15066244	13245504	12.08%	27546660	-82.84%
C7552	12087289	8882203	26.52%	15320250	-26.75%
Average Gain:			13.44%		-37.44%

Source: The author

The table 4.4 shows the accumulated worst distances between inverters and their targets for each inverter tree. In this case we can see that the WorstFirst heuristic was better than random, in spite of being considerably worse at the total inverter-target distance. This is consistent with its concept, which tries to insure the furthest targets get a decent chance at an inverter. However, the K-means heuristic was even better at the worst delay.

Table 4.4 – Summed Worst Inverter Tree distances

Circuit	*Random*	*Kmeans*	*Gain*	*WorstFirst*	*Gain*
C432	155790	152650	2.02%	154425	0.88%
C499	608260	574051	5.62%	590327	2.95%
C880	448967	424502	5.45%	479397	-6.78%
C1355	562710	519378	7.70%	546550	2.87%
C1908	525952	493628	6.15%	519850	1.16%
C2670	652231	596555	8.54%	646941	0.81%
C3540	1605380	1454501	9.40%	1583125	1.39%
C5315	2134803	1894941	11.24%	2101363	1.57%
C6288	3938003	3325698	15.55%	3691227	6.27%
C7552	2021727	1987956	1.67%	1955881	3.26%
Average Gain:			7.33%		1.44%

Source: The author

5 CONCLUSION

This work presents a multi-objective buffer insertion approach to limiting fanout in simple-cells-based circuits taking into consideration the initial positions as well as an I/O timing budget. This approach was based on Matos et al. fanout limiting algorithm, presented in Matos et al. (2013) and Matos (2014), expanding it to include critical path allotment and locality dependent inverter wiring allocation.

The approach was applied to a set of ten combinational circuits (BRGLEZ; FUJIWARA, 1985), and was shown to produce circuits with smaller critical paths when compared to Matos's initial algorithm. Three critical selection and insertion algorithms were developed and compared. The critical selection was found to be crucial for reducing delay by means of heuristically rearranging delays in the inverter trees. Two inverter wiring algorithms were presented and compared. The inverter target delegation defined a decent locality-based wiring, which also served to more accurately estimate the circuit-wide delays. The methodology for applying the buffer insertion throughout the circuit was also discussed and tested. This work has shown a way of taking physical information into consideration during logic synthesis in order to further optimize the circuit topology.

Several modifications could be made to improve this work. The way we currently assess criticality and, consequently, select critical paths may be changed to consider timing slacks. Together with the critical selection algorithms presented herein, we should also consider those paths with worst negative slack as critical (or maybe optimize the total negative slack as well). The initial delay estimation could add one inverter delay to negative consumers. The Inverter Group Difference algorithm could take the inverse polarity targets into consideration. The prune could eventually leave a few more inverters than necessary if it could save a bit of some target's delay. The target pre-delay could be taken into account during buffer insertion. The pre-delay could reveal that some targets depend on a slower signal and therefore its delay could not be improved despite its best efforts. Multiple passes could be done in each node. Every time the buffer insertion runs on a node, its information is updated and the circuit-wide delay estimation is more accurate. This could result in a virtuous cycle of iterative refinement. Hopefully this work was just the first step of many in this direction.

REFERENCES

ALEGRETTI, C. et al. Analytical logical effort formulation for minimum active area under delay constraints. In: **Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.: s.n.], 2013.

BAKOGLU, H. **Circuits, Interconnections, and Packaging for VLSI**. Reading, MA: Addison-Wesley Publishing Company, 1990.

BRGLEZ, F.; FUJIWARA, H. A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran. In: **International Symposium on Circuits and Systems (ISCAS)**. [S.l.: s.n.], 1985.

CALLEGARO, V. et al. A bottom-up disjoint-support decomposition based on Boolean difference analysis. In: **International Workshop on Logic and Synthesis (IWLS)**. [S.l.: s.n.], 2015.

DARRINGER, J. A. et al. Logic synthesis through local transformations. **IBM Journal of Research and Development**, v. 25, n. 4, p. 272–280, 1981.

KRAUSE, E. F. **Taxicab geometry: an adventure in non-Euclidean geometry**. New York: Dover Publ., 1987.

MACHADO, L. et al. Logic synthesis for manufacturability considering regularity and lithography printability. In: **International Symposium on VLSI (ISVLSI)**. [S.l.: s.n.], 2013.

MACHADO, L. et al. KL-cut based digital circuit remapping. In: **NORCHIP**. [S.l.: s.n.], 2012.

MATOS, J. M. **Graph-Based Algorithms for Transistor Count Minimization in VLSI Circuit EDA Tools**. Dissertation (Master) — PGMicro/UFRGS, March 2014.

MATOS, J. M. et al. A Benchmark Suite to Jointly Consider Logic Synthesis and Physical Design. In: **International Symposium on Physical Design (ISPD)**. [S.l.: s.n.], 2015.

MATOS, J. M.; REIS, A. Placed AIGs as a Logical-Physical Data-Structure for VLSI Circuit Design. In: **International Workshop on Logic and Synthesis (IWLS)**. [S.l.: s.n.], 2015.

MATOS, J. M. et al. Deriving Reduced Transistor Count Circuits from AIGs. In: **Symposium on Integrated Circuits and Systems Design**. [S.l.: s.n.], 2013.

MOORE, G. Cramming More Components onto Integrated Circuits. **Electronics**, p. 82–85, 1965.

NEUTZLING, A. et al. Threshold Logic Synthesis Based on Cut Pruning. In: **International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2015.

POSSANI, V. et al. Towards Optimal Area Synthesis of Small Combinational Circuits. In: **International Workshop on Logic and Synthesis (IWLS)**. [S.l.: s.n.], 2015.

WAGNER, F. R.; REIS, A. I.; RIBAS, R. P. **Fundamentos de circuitos digitais**. [S.l.]: Sagra Luzzatto, Porto Alegre, 2006.