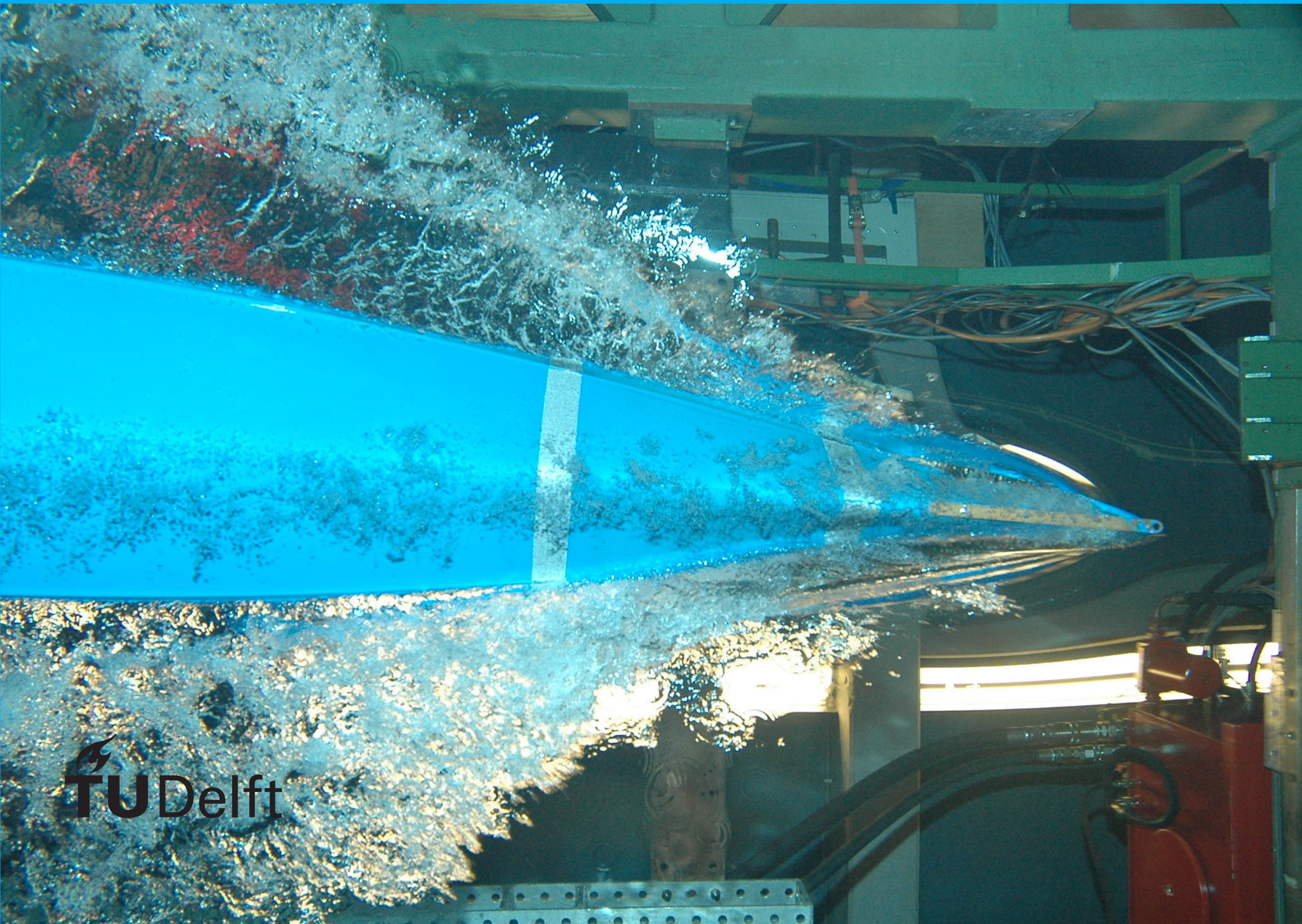
Quality and Cost Modeling Of 3D Stacked ICs

Michael Mainemer



Quality and Cost Modeling Of 3D Stacked ICs

by

Michael Mainemer

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday April 30, 2018 at 12:00 PM.

Student number:	4508572	
Project duration:	May 1, 2017 – April 30, 2018	
Thesis committee:	Prof. dr. ir. S. Hamdioui,	TU Delft, supervisor
	Dr. Ir. A. Bossche ,	TU Delft
	Dr. Ir. M. Taouil ,	TU Delft

TU Delft

Acknowledgments

I would like to thank my supervisors Prof. Said Hamdioui and Dr. Ir. M. Taouil for all their assistance and support during the completion of this thesis. I also want to express my gratitude to PDEng. MSc. Erik Jan Marinissen from IMEC and TU Eindhoven for all his advice, which allowed me to better understand and execute this project.

Finally, I want to thank my family, girlfriend and friends for all their support. Your unconditional love and constant encouragement, is what has allowed me to conclude this part of the journey.

Michael Mainemer
Delft, April 2018

Contents

List of Figures vii

List of Tables ix

Abbreviations xi

1 Introduction 3

1.1 Motivation . . . 3
1.2 State of the Art . . . 5
1.3 Contribution . . . 6
1.4 Thesis Organization . . . 7

2 3D Stacked ICs 9

2.1 3D Die Stacking . . . 9
2.1.1 Stacking Process . . . 11
2.1.2 Types of stacking . . . 13
2.2 Stacked ICs Architectures . . . 14
2.3 Challenges . . . 15
2.4 Opportunities . . . 17

3 Cost Modeling 19

3.1 Classification . . . 19
3.2 3D COSTAR . . . 22
3.2.1 Tool flow . . . 25
3.2.2 Sensitivity Analysis . . . 26
3.3 Limitations . . . 27

4 Extensions to 3D COSTAR 29

4.1 Test Flow Optimization . . . 29
4.1.1 Shortest Path Problem . . . 31
4.1.2 Modified Dijkstra . . . 35
4.2 Stacking Order Analysis . . . 40
4.3 TSV Manufacturing Cost . . . 42

5 Simulation Results 43

5.1 Implementation . . . 43
5.1.1 Test Flow Optimization . . . 45
5.1.2 Stacking Order Analysis . . . 49
5.1.3 TSV Manufacturing Cost . . . 51
5.2 Case studies . . . 51
5.2.1 Case Study 1, FPGA 2.5D stack . . . 52
5.2.2 Case Study 2, Memory 3D Stack . . . 55
5.2.3 Case Study 3, 5.5D Stack . . . 57

6 Conclusion 61

6.1 Summary . . . 61
6.2 Future Work . . . 61

Bibliography 63

List of Figures

1.1 Moore's law evolution [55] . . . 3
1.2 Different functions on one 3D SIC [23] . . . 4
1.3 Classification of existing cost models for the stacked IC technology . . . 6

2.1 IC Packaging Evolution [2] . . . 10
2.2 3D Packaging [1] . . . 10
2.3 3D Die Stacking [1] . . . 11
2.4 Monolithic [37] . . . 11
2.5 Stacking process [1] . . . 12
2.6 TSV type depending on the moment of fabrication [3] . . . 12
2.7 Thinning process [32] . . . 13
2.8 Bonding methods: direct oxide bonding, adhesive bonding and direct metal bonding [26] . . . 13
2.9 Types of Stacking [19] . . . 14
2.10 Stacking Architectures [34] . . . 15
2.11 Common types of defects on stacking process [12] . . . 16
2.12 Typical test moments on a 2D and on a 3D flow [46] . . . 16

3.1 Existing Cost Models Classification . . . 20
3.2 Cost Classes . . . 22
3.3 a) Wafer processing [56]. b) Stacking process [54] . . . 23
3.4 Test flow for a SIC [46] . . . 24
3.5 Logistics costs [52] . . . 24
3.6 3D COSTAR flow [48] . . . 25
3.7 Stack representation in 3D COSTAR [51] . . . 26
3.8 Main case and sensitivity analysis . . . 26

4.1 Conformation of a Test Flow . . . 30
4.2 Depiction of test flow problem through a tree . . . 30
4.3 Shortest Path Classification . . . 31
4.4 Tree Structure used in Examples . . . 32
4.5 Breadth First Search . . . 33
4.6 Depth First Search . . . 34
4.7 Dijkstra's Algorithm . . . 35
4.8 Test moments explored in example . . . 37
4.9 Test moments explored in example . . . 37
4.10 Values used for the nodes of the second level of the tree . . . 37
4.11 Values used for the nodes of the third level of the tree . . . 38
4.12 First step: exploring the children of the original node . . . 38
4.13 Second step: exploring node 1's children . . . 39
4.14 Third step: exploring node 4's children . . . 39
4.15 Final step: Finding the cheapest solution . . . 40
4.16 Stacking order, first case . . . 41
4.17 Stacking order, reordered . . . 41
4.18 Separation of wafer processing costs . . . 42

5.1 Files conforming 3D COSTAR . . . 44
5.2 Test Flow Optimization Flowchart . . . 45
5.3 Test moments explored in example . . . 48
5.4 Example results of a test flow optimization case with equal cost . . . 48

5.5 Stacking order determination . . . 50
5.6 Cost of different stacking orders FPGA case . . . 53
5.7 Cost and escapes for optimization for lower cost in FPGA case . . . 54
5.8 Number of cases explored to find cost solution in FPGA case . . . 54
5.9 Cost and escapes for optimization for lower escapes in FPGA case . . . 54
5.10 Number of cases explored to find escapes solution in FPGA case . . . 55
5.11 Cost of different stacking orders memory case . . . 56
5.12 Cost and escapes for optimization for lower cost in memory case . . . 57
5.13 Number of cases explored to find cost solution in memory case . . . 57
5.14 Cost and escapes for optimization for lower escapes in memory case . . . 58
5.15 Number of cases explored to find escapes solution in memory case . . . 58
5.16 Cost and escapes for optimization for lower cost in 5.5D case . . . 59
5.17 Number of cases explored to find escapes solution in 5.5D case . . . 59
5.18 Cost and escapes for optimization for lower cost in 5.5D case . . . 60
5.19 Number of cases explored to find escapes solution in 5.5D case . . . 60

List of Tables

4.1 Total distances for example case 32
4.2 Comparison of analyzed algorithms 36

5.1 Parameters for FPGA 2.5D case 52
5.2 FPGA case stacking order results 53
5.3 Results of optimizing for lower cost in FPGA case 53
5.4 Results of optimizing for lower escapes in FPGA case 55
5.5 Parameters for Memory 3D case 56
5.6 Memory case stacking order results 56
5.7 Results of optimizing for lower cost in memory case 57
5.8 Results of optimizing for lower escapes in memory case 58
5.9 Results of optimizing for lower cost in 5.5D case 59
5.10 Results of optimizing for lower escapes in 5.5D case 60

Abbreviations

B2F	-	Back to Face
BEOL	-	Back End Of Line
BGA	-	Ball Grid Array
3D COSTAR	-	Third Dimension Cost Architecture
CPU	-	Central Processing Unit
D2D	-	Die to Die
D2W	-	Die to Wafer
DfT	-	Design for Testability
DPPM	-	Defective Parts Per Million
DRIE	-	Deep Reactive Ion Etching
F2F	-	Face to Face
FC	-	Fault Coverage
FEOL	-	Front End Of Line
FIFO	-	First In First Out
FPGA	-	Field-Programmable Gate Array
GPU	-	Graphics Processing Unit
IC	-	Integrated Circuit
I/O	-	Input/Output
KGD	-	Known Good Die
MIV	-	Monolithic Inter-tier Via
PCB	-	Printed Circuit Board
PoP	-	Package on Package
PPM	-	Parts Per Million
SIC	-	Stacked Integrated Circuit
SiP	-	System in Package
TMAH	-	Tetramethylammonium hydroxide
TPV	-	Through Package Via
TSV	-	Through Silicon Via
W2W	-	Wafer to Wafer

Abstract

Since the 1960s, the semiconductor industry's rate of progress has followed by what is commonly known as Moore's law. In recent years, the traditional approach of downscaling the feature size and increasing the operating frequency of the devices to improve the performance is slowing down due to the reaching of physical limits. Different solutions have been proposed to increase the performance such as using multi-core platforms, able to work in parallel. However, this introduced other issues such as a high latency and power dissipation due to the long communication distances. To overcome this, 3D Stacked ICs have been proposed, allowing for shorter vertical interconnects and higher transistor density. With the introduction of 3D stacked ICs (3D-SIC), the manufacturing processes and its test flow become more complex. As each chip requires to be tested prior to deployment, its crucial to find optimal test flows..

Developing an effective test strategy for 3D SICs is a complex task. During a 2D process, there are typically two test moments (i.e., wafer test and packaging test). However for a 3D process, the number of possible moments grows based on the number of dies in the stack as many partial stacks test can be performed. In addition, also the interconnects between the stacked dies can be tested, as they play a crucial role in determining the correct functioning of the device. Testing such a partial stack may reduce the cost as it prevents good dies being stacked on them, but on the other hand may increase the total cost due to extra test costs. To help making such decisions, appropriate cost models are required. They are tools used to estimate the cost of a test flow typically applied at an early design stage. The information obtained from them can then be used to optimize the process in question and reduce the cost.

In this thesis, the functionality of 3D COSTAR is extended. 3D COSTAR is a tool developed with the goal of modeling the complete 3D SIC supply chain, including design, manufacturing, test, packaging and logistics. This thesis first presents a test flow optimization algorithm. It is used to evaluate and find the best test strategy, i.e., the highest fault coverage at the lowest cost. Second, a stacking order analysis is performed in which the impact of different stacking orders is evaluated. The goal is to evaluate whether a different stacking order reduces the overall cost. We evaluated both extensions using three case studies. Overall, lower cost solutions were found across all three cases. With respect to test flow optimization, cost reductions of 4%, 6% and 27% were obtained for the three case studies. With respect to stacking order analysis, a cost improvement up to 16% was realized.

1 Introduction

This chapter serves as a brief introduction to the topics explored in this thesis. It highlights the motivation for moving from 2D ICs to 3D stacked ICs, as well as the need for test flow modeling. Section 1.1 motivates the need for cost modeling and test flow optimization for 3D stacked ICs. Section 1.2 discusses the state-of-the-art in cost modeling the stacked IC process. Section 1.3 discusses the contributions of this thesis. Finally, Section 1.4 presents the thesis outline.

1.1. Motivation

Since announced in the 60s by Gordon Moore, Moore's law has set the pace for technological innovation in the semiconductor industry, where the majority of efforts made by companies have focused on packing more computing power in a smaller area by shrinking the feature size as manufacturing technology evolves [35].

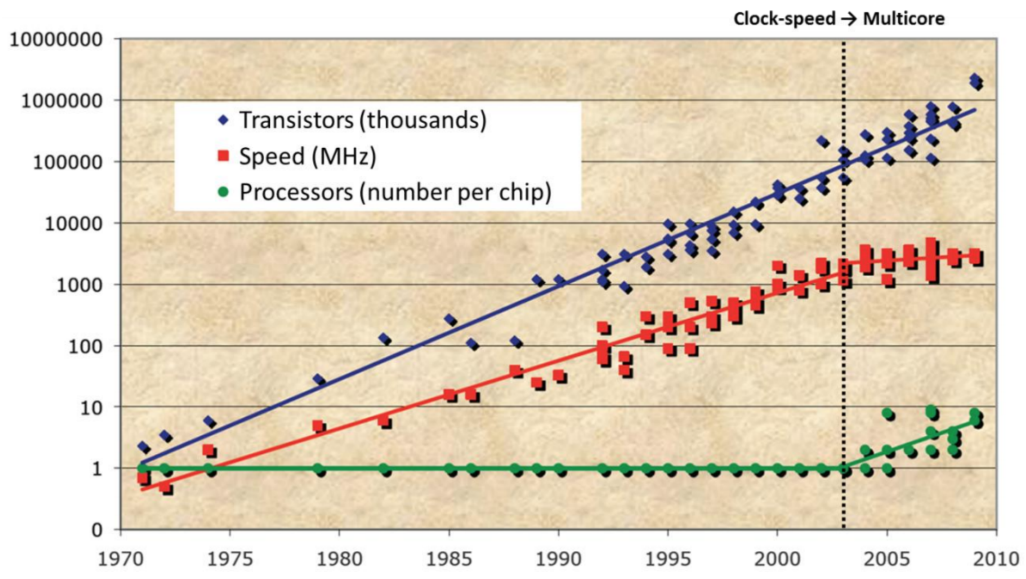


Figure 1.1: Moore's law evolution [55]

As the feature size is slowly reaching its physical limits, maintaining the down scaling becomes challenging. Such challenges are an increase in current leakage due to direct tunneling, challenges to adapt the lithographic process to keep up with transistor size and finally the physical limit to scale beyond the size of atom [27]. They motivated scientists and engineers to explore non-traditional ways of increasing the number of transistors, as the shrinking of feature size becomes more difficult. Of great importance to this diversification process of was when designers noticed that they had reached the power wall [39] around 2004. After this moment, it was not possible to keep lowering the operating voltage and increasing the CPU frequency. As a solution, the multi-core CPU era began. Using multicores, it is possible to continue increasing the processing power by executing tasks in parallel on different cores; instead of relying a faster execution per task using improved technology, tasks would be executed in parallel to increase the performance. In case a core is not used, it can be turned off. However, also here new challenges arose.

In case multiple cores are placed on a single 2D chip, the distance required to send information between them (or latency) becomes longer, thus making the operation slower. At the same time, as interconnects get longer, the power consumption also increases. Two reasons for this are the inherent resistance of the wires, as well as the need of repeaters to boost the signal [38]. The current trend of packing more functionality into electronic chips while making them smaller has lead to new methods to physically organize ICs, i.e., using the vertical dimension to stack them. 3D IC manufacturing makes it possible to solve or alleviate these problems and gives an opportunity to continue increasing the capabilities of ICs [50].

Part of the appeal for 3D ICs comes from the fact that different functional chips can be stacked on top of each other possibly with different technologies, while minimizing the footprint. This saves money as smaller Printed Circuit Boards can be used. In addition, new features can be added vertically in the stack which otherwise would not have fit in it.

Another important advantage comes from the manufacturing process. For a given defect density, smaller dies have a higher yield than larger dies [36]. As die area increases, the yield decreases due to a higher probability that a defect is present somewhere along the die, so to prevent this from happening, large dies can be partitioned into smaller dies, increasing the yield of the manufacturing process and reducing costs.

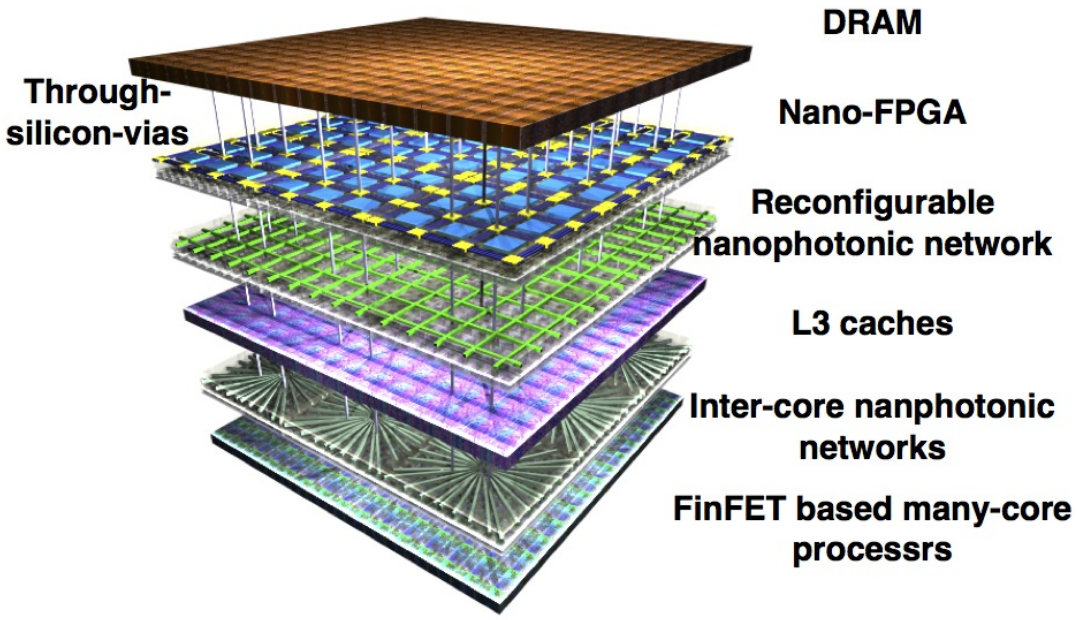

DRAM
Through-silicon-vias
Nano-FPGA
Reconfigurable nanophotonic network
L3 caches
Inter-core nanphotonic networks
FinFET based many-core processrs

Figure 1.2: Different functions on one 3D SIC [23]

All these advantages come with a price. As the devices become more complex, with more dies in a single package, the complexity of the manufacturing process also grows. Next to the manufacturing process, another very important task is the testing process, which ensures the quality of a manufacturing process, by detecting defective chips before being sent to the customer. In the same way that the manufacturing complexity grows with the number of dies in the 3D ICs, the testing process also becomes more complicated, having to consider a very large number of possibilities regarding the best strategy to apply the tests.

To carry out these processes, it is crucial to consider the cost that they entail. A more complex manufacturing process means a higher probability of damaging the devices, as well as a more difficult test, to cover all the possible defects introduced by stacking the devices. It is for this reason that this project has been proposed, to provide a tool capable of assisting during the planning and decision making stages of a stacked IC process. This project focuses on the test flow and cost modeling and will use as a baseline the model and tool presented in [51] referred to as 3D COSTAR.

1.2. State of the Art

Currently, nine cost modeling tools for 3D Stacked ICs have been presented. They are classified in Figure 1.3. The models fall into different categories and only a very few can be considered as providing a complete solution to this problem. It classifies them based on the used 3D technology (i.e., 3D packaging, Stacked ICs, Monolithic ICs) and the phases of the supply chain they cover (Design, Manufacturing, Test, Packaging and Logistics). These processes are modeled differently in each of them. In some cases, detailed models are used, while in others, only the basic components are considered.

The first model was proposed by Bauer in 2008 [10], focusing on 3D Packaging technology. The model considers packaged devices stacked on top of each other. The model utilizes a simple cost model formulation, with the intention of evaluating the cost of manufacturing a 3D IC and briefly explores how the involved logistics affect the cost.

The introduction of 3D Stacked ICs (3D-SIC), where dies are stacked directly on top of each other interconnected by TSVs, further increases the performance with respect to 3D packaging technologies such as SiP and Pop, as a higher communication bandwidth with a lower power consumption can be realized between the dies with a smaller form factor [8, 40]. Most of the existing models are based on 3D-SICs.

Stow provided in 2016 [44] a cost model that covers the manufacturing and packaging phases with a focus on detailed modeling of the thermal dissipation. Four models have been proposed that additionally consider the test phase. Sengupta in 2011 [42] seeks to find the best test strategy based on the shortest time and lowest number of dies manufactured to meet the desired stacks number. This model's analysis in terms of manufacturing and packaging is not as thorough as the others, but the author introduces another dimension by comparing three different testing scenarios. Another model which also covers the same classes was proposed by Chou in 2012 [16]. A tool is created based on the model and besides the cost calculation, it is capable of automatically modifying the testing parameters to evaluate different test flows.

The final two models from this group were proposed by Agrawal and Chakrabarty. In the first one [5], their model utilizes a method based on matrix partitioning to find a test flow that minimizes the process' total cost. In the second study [6], they modify their approach for the test flow determination and reformulate it into a shortest path problem, solving it by using the A* algorithm.

Based on the same 3D technology, but now focusing on integral models which cover the complete supply chain, two alternatives are identified. Gruenewald in 2015 [22], developed a cost model for all five phases. As in all previous cases, the model formulation has its own distinct approach, but the fact that all phases are part of it, is noteworthy. A tool is also developed, but it does not provide any comparison or analysis capabilities besides the cost calculation and its breakdown.

The final cost model pertaining to the Stacked ICs technology is 3D COSTAR, by M.Taouil [47–49, 52]. This model encompasses all five cost classes, providing detailed analysis based on its highly comprehensive input. The elements involved in the process, are represented by detailed parameters, which produce a complete analysis. Additionally, a tool has been developed based on the cost model, the tool is also capable of performing a sensitivity analysis, introducing a degree of comparison capabilities.

The last study concerns the Monolithic 3D SIC technology. It was published in 2016 by Gitlin [21] and it introduces a basic way of modeling the manufacturing cost by relating it with the cost of manufacturing the same device on a 2D version.

The existing options offer interesting possibilities regarding Another weak spot detected during the research is that most of the existing solutions do not offer the possibility to automatically generate cases and compare them to find a better solution and the ones that do have some kind of search space generation, are lacking in some other way.

After analyzing the existing work, it is evident that there is no solution that covers the whole supply chain and provides a comprehensive enough analysis. Based on its shortcomings and strengths, this work aims to extend the capabilities of 3D COSTAR through the incorporation of additional analysis methodologies, transforming it from a cost modeling tool, to one capable of performing a more comprehensive analysis.

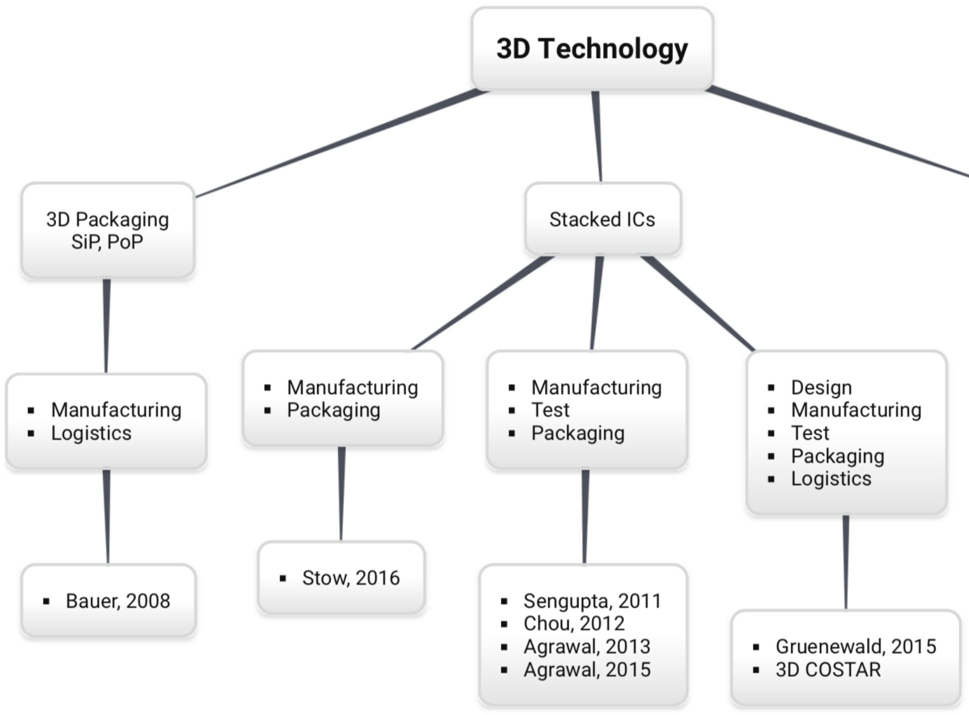

3D Technology
3D Packaging SiP, PoP
Stacked ICs
Monolithic
▪ Manufacturing ▪ Logistics
▪ Manufacturing ▪ Packaging
▪ Manufacturing ▪ Test ▪ Packaging
▪ Design ▪ Manufacturing ▪ Test ▪ Packaging ▪ Logistics
Manufacturing
▪ Bauer, 2008
▪ Stow, 2016
▪ Sengupta, 2011 ▪ Chou, 2012 ▪ Agrawal, 2013 ▪ Agrawal, 2015
▪ Gruenewald, 2015 ▪ 3D COSTAR
▪ Gitlin, 2016

Figure 1.3: Classification of existing cost models for the stacked IC technology

1.3. Contribution

The work done in this project focuses on extending the functionality of 3D COSTAR. Different tasks have been performed. First, the existing code has been revised. Second, multiple functions have been re-implemented and the output has been improved. Thirdly, the cost of the manufacturing of TSVs has been included in the model. Finally, two optimization techniques have been added to the tool. The main contributions of this thesis are:

- **Code revamping:** functions from the original version of the tool have been rewritten in order to improve code readability, performance and produced output results. In addition, part of the structure has been modified to accommodate the new functionality.

- **The addition of the TSV manufacturing cost:** the cost modeling of the manufacturing cost of TSVs has been added to the tool. This allows to differentiate between the TSV and die cost of pre-bond dies.

- **Demonstration and validation of test flow optimization:** based on a few input parameters that the user need to specify, the tool is able to generate different test cases and automatically explore them by comparing. Due to the possibility of having a very large search space, a shortest path solution has been implemented. It calculates the best solution regarding cost and quality for the generated cases. This analysis works through a modified version of Dijkstra's algorithm, capable of guiding its search to find the correct result in the most efficient way.

- **Demonstration and validation of stacking order optimization:** here, the stacking order is modified and different alternatives are tried exhaustively out to find the most cost-efficient implementation. This might give designers the option to rethink about the stacking flow in case huge cost benefits can be realized.

1.4. Thesis Organization

The rest of this thesis is organized in five chapters. The next two chapters provide background knowledge related to 3D SICs. The subsequent chapter provides a detailed description of the extension at high level. The chapter thereafter takes a deeper look into the implementation details and applies the tool to three different case studies, and the last chapter contains the conclusion. Below the organization of each chapter is provided.

Chapter 2: it explores the basics behind the 3D stacked IC process.

Chapter 3: it analyses the existing cost models, and highlights their strengths and weaknesses. 3D COSTAR is explained in more detail as the new functionality is added to this tool.

Chapter 4: it discusses on a high level the proposed extensions to the tool.

Chapter 5: it explains the implementation details of the extensions and validates the results utilizing three case studies.

Chapter 6: it provides a summary of this thesis as well as potential future work.

2 3D Stacked ICs

This chapter presents the 3D stacked IC, it discusses the different technologies, as well as the manufacturing steps involved in the stacking process. Section 2.1 highlights the 3D technologies and the process to manufacture them. Section 2.2 discusses the different possible architectures. Section 2.3 focuses on the challenges faced by this technology and Section 2.4 on the opportunities.

2.1. 3D Die Stacking

Since the birth of the Integrated Circuits industry, one of the fields where more work has been dedicated to is to achieve higher integration in and between ICs. To achieve better integration, many efforts have been made on different areas of the IC production process. One of the main pillars for this has been the continuous evolution of the technology node reduction, which produces smaller devices. The device's architecture also dictates in great form different possibilities in terms of how to pack more functions together. Also of importance are the packaging changes that accompany the improvements to the ICs.

Related to the reduction in transistor size, the first steps taken in terms of packaging where to develop different package types. This provided gains in area by packaging more tightly the devices and rearranging the pins to achieve reductions in the pitch. As area and performance requirements kept pushing for a tighter integration, new solutions were adopted as Multi-Chip Modules which later gave place to System-on-Chip devices.

Based on the great difficulty to keep shrinking the devices and shortening the interconnect, a different approach was proposed which gave birth to the stacked IC solution.

The stacked IC solution, typically called 3D SIC can be divided into three main groups, 3D Packaging, 3D Die Stacking and 3D Monolithic. These three groups refer to the level of integration achieved, which is a product of the followed manufacturing process and which will be discussed next.

3D Packaging

3D packaging consists of multiple dies stacked vertically at packaging level. This means, that each die goes individually through the majority of the manufacturing process without major influences of the other dies that will form the stack. Near the end of the process, the dies are prepared for the stacking operation, where the interconnect between dies can be done by wire bonding, BGA stacking or Flip Chip. An advantage of this method lies in its simplicity compared to the others to achieve the 3D structure. It is more simple because there are no new structures needed just the way of organizing them. Mainly, there are two ways of packaging ICs that conform with the 3D Packaging class:

- **System in Package (SiP):** Consists of naked chips stacked on top of each other and a substrate, to which they connect through wires. As can be seen on the leftmost device depicted in Figure 2.2, although both dies are packaged together, their connection with external parts is done without going through other dies part of the stack.

- **Package on Package (PoP):** In this case, fully packaged chips are stacked on top of each other. For this method to work, more provisions need to be taken care of concerning the relation between packages. This because as depicted on the rightmost device from Figure 2.2, it is necessary to have interconnect

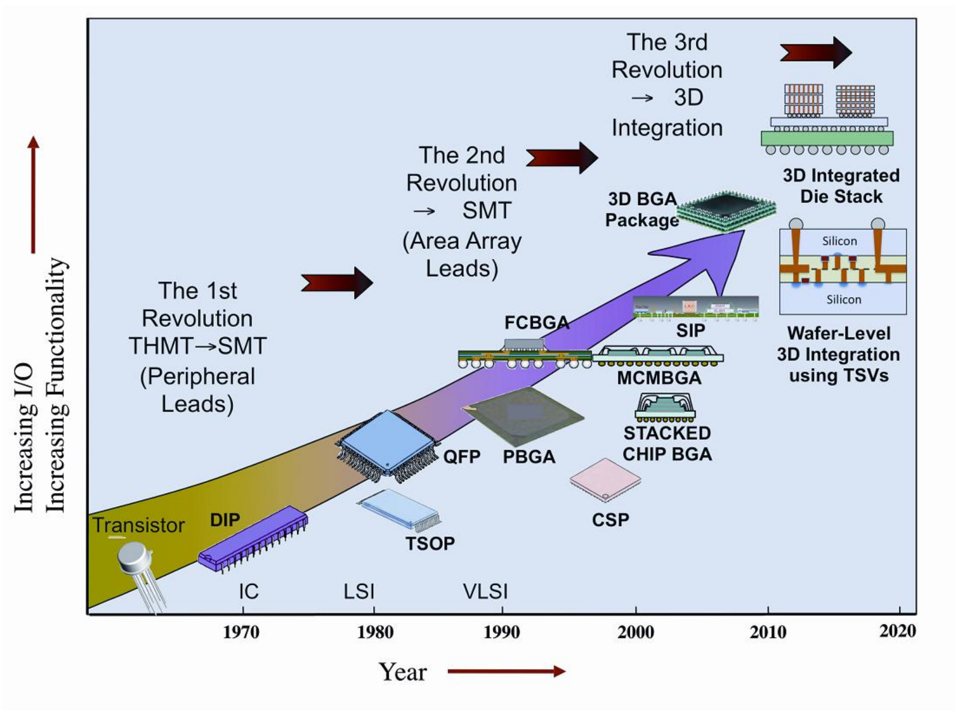


Figure 2.1: IC Packaging Evolution [2]

structures that enable the signal propagation from packages upper in the stack all the way down to the substrate of the stack that will have a structure to make contact with the PCB. Also, in case of communication between stacked elements, it will have to travel all the way from the originating element, to the substrate and from there to the destination.

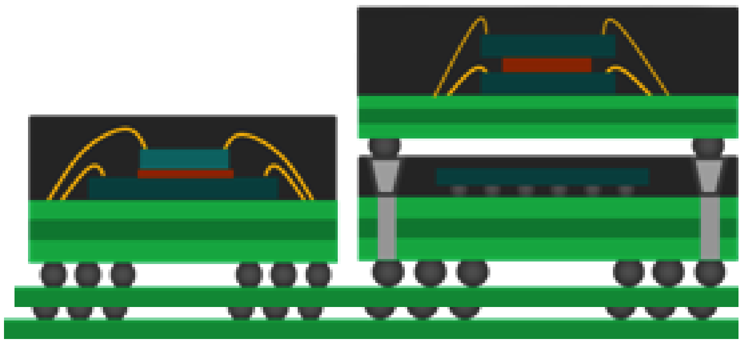


Figure 2.2: 3D Packaging [1]

3D Die Stacking

The dies that will conform the stack go through the die fabrication process individually, after this, they are bonded together. For the communication between dies and the package substrate, two structures are used. The first one is Through Silicon Vias (TSV), for cases where there is no contact between metal layers of the dies as in Face to Back stacking and is required to go through the substrate. The second option is microbumps, which are used to create the contacts between each die's pads to the other[31]. Figure 2.3 shows the configuration of a stack based on the 3D Die Stacking method. To be able to reach each other, it is necessary to go through the active layers and the die's substrate, so the TSVs are shown as vertical white lines with a red surrounding and interconnected with small black circles that represent the microbumps. This method although more complex to fabricate due to the addition of TSVs and microbumps, has some benefits over 3D Packaging, mostly seen in the shorter interconnect and savings in vertical area[8, 40].

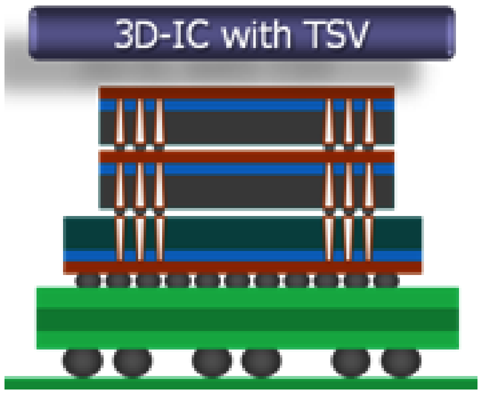


Figure 2.3: 3D Die Stacking [1]

3D Monolithic

In this technology, two or more active silicon layers with devices are fabricated sequentially one on top of the other. The devices are interconnected with more advanced vias called monolithic inter-tier vias (MIV)[28]. To prevent shorting between active layers, dielectric material is placed between them . By having the devices located at such proximity, the delays and signal loss are reduced to a minimum, improving performance and power consumption. Physical advantages are also notorious due to a much higher integration density, allowing to save in silicon and space[4]. Although very promising in terms of performance improvement, it is not a commercially exploited technology due to the fabrication complexity. One way this manifests is in thermal issues caused by limited access to dissipate heat in densely concentrated active regions[45].

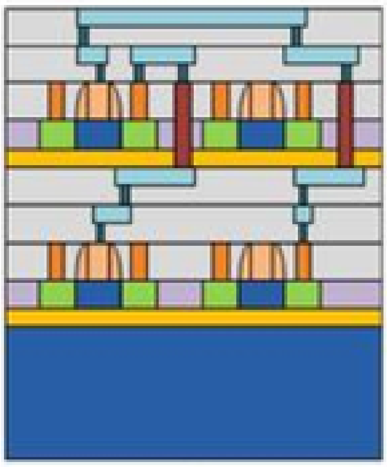


Figure 2.4: Monolithic [37]

2.1.1. Stacking Process

The stacking process consists of all the steps that need to be taken to prepare the dies to be stacked. As will be discussed later, the stacking can be performed on three different ways, Wafer to Wafer (W2W), Die to Wafer (D2W) and Die to Die (D2D).

Independent from the way that they will be stacked, wafers and dies need to go through special manufacturing processes to enable the 3D integration. These three processes are:

- **Through Silicon Vias fabrication:** This step is concerned with the manufacturing of the TSVs. TSVs can be of different materials such as Copper or Tungsten and the material and size will be heavily influenced

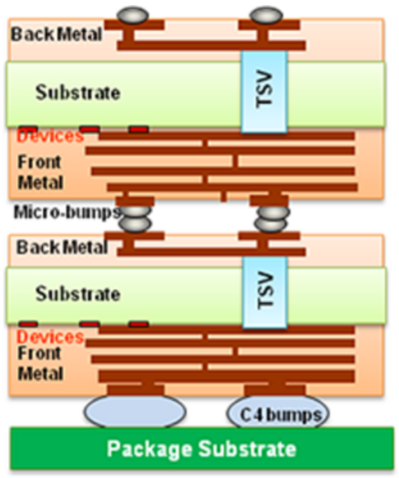


Figure 2.5: Stacking process [1]

by the moment in the manufacturing process in which they are created. They can be fabricated either in the 2D stage of the process, or during the 3D stage. For the former, there are two cases, via-first before FEOL and via-middle between FEOL and BEOL[20]. For the latter, it is called via-last which can be done before or after thinning [24]. Typical diameter dimensions for via-first are between 1 and 10 μm and 3:1 to 10:1 aspect ratios. Via-first also has the particularity that if something goes wrong, not much work has been invested on that particular wafer. For via-last, typical diameter dimensions are in the 10 to 50 μm range with aspect ratios of 3:1 to 15:1, being capable of handling more current thus used for power distribution[38]. The drilling task in the TSV fabrication can only be done by two different methods, this because of the extreme aspect ratios which the TSVs are required to have. The two methods being laser drilling or deep reactive ion etching (DRIE), with DRIE being a more mature method, thus more prevalent in industry[11].

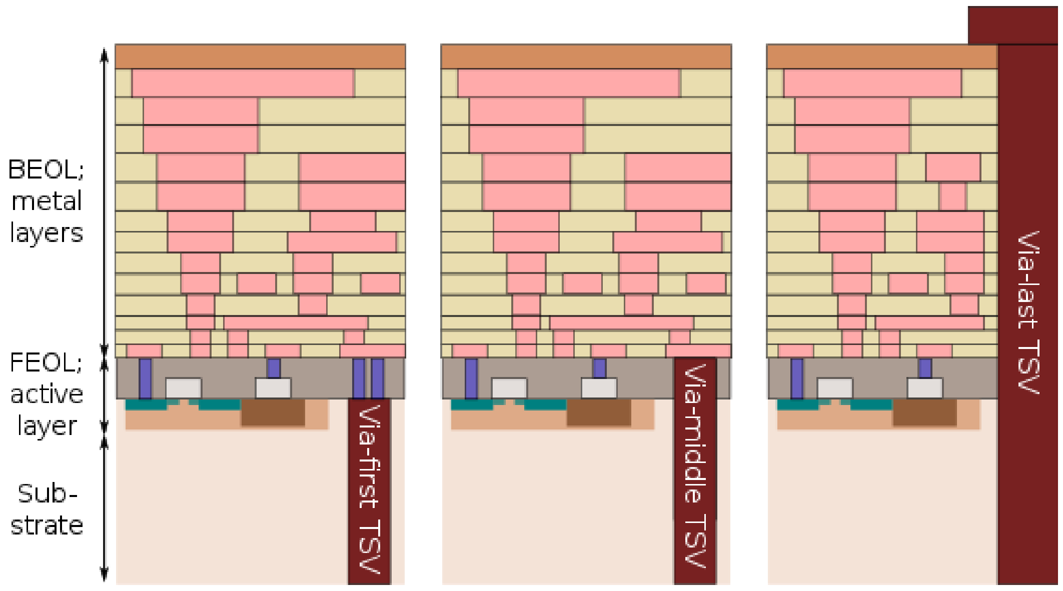


Figure 2.6: TSV type depending on the moment of fabrication [3]

- **Thinning:** With typical wafer thickness of 750 μm and TSVs in many cases with less than 100 μm of height, it is necessary to reduce the thickness of the wafer, so the vias can protrude enough to make contact with other elements. The thinning process is done by a three step process: grinding, polishing and wet etching. Grinding is done first to significantly reduce the thickness of the wafer. Once the wafer is approaching the desired thickness, polishing takes place to remove imperfections and to prevent damaging important structures as the TSVs. Finally wet etching is done via tetramethylammonium hydroxide (TMAH). This removes the last excess of materials and leaves the tips of the TSVs sticking out[30]. Due to the high mechanical stress applied to the wafer throughout this process, the wafer is temporarily bonded to a carrier wafer that will prevent it from breaking and when the thinning is done the carrier is debonded.

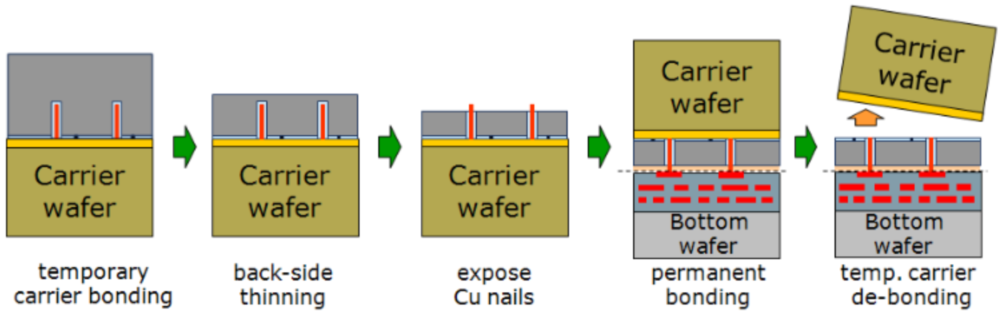


Figure 2.7: Thinning process [32]

- **Bonding:** The last process that needs to be discussed is bonding. It can be temporary, to assist in specific tasks of the fabrication process, as in the thinning process, or permanent, which is the bonding type used to adhere the different layers that form the stack. There are three main methods used for bonding: adhesive bonding, direct oxide bonding (SiO_2/SiO_2) and direct metal bonding (Cu-Cu, Au/Au, Cu-Sn/Cu-Sn). For all three methods it is necessary to raise the temperature[29]. Besides joining the different layers, another important element of the bonding process is ensuring the connection between them. Although different alternatives have been explored, the prevalent way for ensuring the connectivity is by microbumps. Microbumps are made of conductive metallic elements as in Cu, Ni, SnAg, with diameters varying between 15 to 25μm[18]. They are placed on pads or at the end of the TSVs aligned with each other so that when heat is applied, a connection will be made between dies[33]. Although the goal of the bonding process is always to stack together a number of dies, it can vary depending on how those dies are brought into the process. There are three possible approaches which will be discussed next.

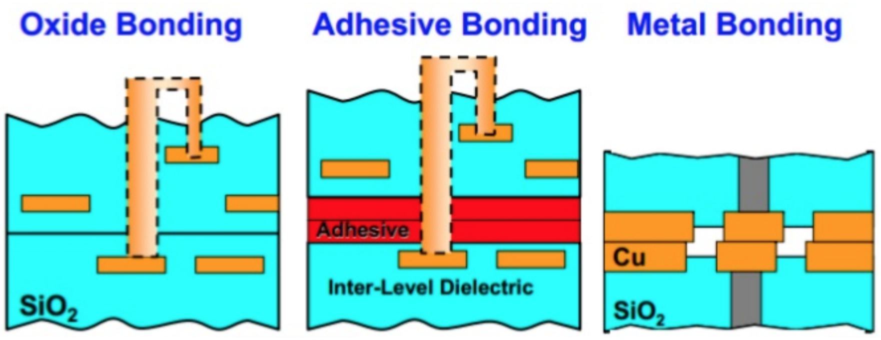


Figure 2.8: Bonding methods: direct oxide bonding, adhesive bonding and direct metal bonding [26]

2.1.2. Types of stacking

As it was mentioned before, the stacking can be done in three different ways, each with advantages and disadvantages compared to the others.

- **Wafer to Wafer (W2W):** This approach consists of stacking complete wafers on top of other wafers and once bonded, they go through the dicing process. The main advantage of this method is the high throughput as many dies get stacked at the same time in one operation, as well as having the easiest alignment of all three. Some disadvantages are that it is only possible for dies with the same size; lowest compound yield due to impossibility to match good dies and leave bad dies from stacking. Stacks can be damaged during dicing.

- **Die to Wafer (D2W):** In this case, a wafer will be stacked with individual dies, which can be selected through testing to only include known good dies (KGD), thus increasing the compound yield. After stacking, the wafer will be diced, but the risks of damaging are lower as only one layer will be cut. This alternative has lower throughput than W2W because the alignment is more complex, but this can be balanced with higher yield due to selection of good dies and also the ability of using different sized dies, as long as the wafer elements are larger.
- **Die to Die (D2D):** The last method consists of diced KGDs, which are individually aligned one on top of the other and bonded together. D2D can achieve the highest compound yield of all three methods by eliminating from the stack, the risks of damaging it due to the dicing process. As in D2W, the alignment operation is very complex, being this its main disadvantage.

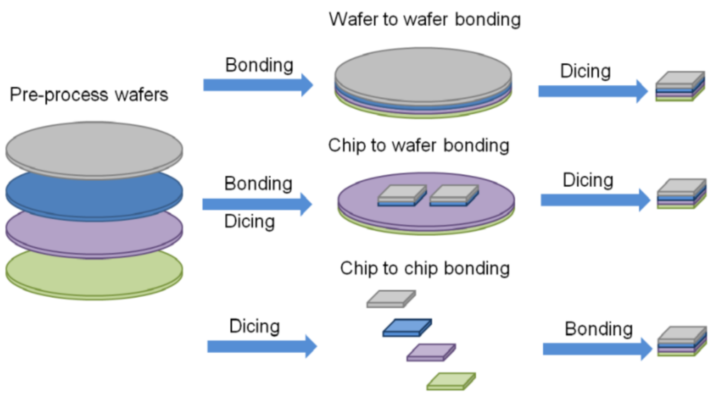


Figure 2.9: Types of Stacking [19]

2.2. Stacked ICs Architectures

Once a stacked IC solution is sought, different options exist, providing different advantages but with the cost of more complexity in terms of the design and manufacturing process, which goes closely together with testing.

- **2.5D:** The 2.5D stacked IC does not stack dies on top of each other. The dies are placed in a 2D formation, with all off these dies stacked on top of a silicon interposer. The interposer is in charge of providing interconnect channels between the dies by a combination of TSVs and traces and to connect the dies to the package substrate, through which the dies will connect with external components in the PCB. Because the interposer uses a similar fabrication process as the dies, the interconnect can achieve sizes similar to the ones on the dies, which is orders of magnitude smaller than what is possible on a PCB or other existing alternatives. In essence, the interposer is a die with the metal layers and TSVs. By not having active dies on top of each other, the way to dissipate heat remains similar to that on a 2D architecture, while having a much smaller footprint, making this architecture attractive for high performance computing.
- **3D:** The next approach for organizing a stacked IC is the 3D configuration. In the 3D stacked ICs, the active dies are stacked on top of each other, reducing the footprint of the device to the minimum. By stacking the dies, the traditional way of having each die connected to the substrate is not possible to achieve, therefore access to the dies not directly on top of the substrate is accomplished through TSVs. The vias are located on dies where signals must traverse through them to reach the substrate or other dies with complementary circuitry. As can be seen in Figure 2.10 b, dies 1 and 2 have TSVs going through them, while die 3 does not require them, because it can connect with die 2 without having to go through the die. It uses the regular metal layers of interconnect to guide signals to the pads on the surface and there, with microbumps it passes them to the other die. An important advantage is that a more direct path exists between the dies, thus reducing interconnect length. 3D stacked ICs are more prone to suffer from heat dissipation issues, hence their applications favor the mobile industry, where its advantage of small footprints is a must and due to the reliance on batteries, low power is crucial.

- **5.5D:** This configuration mixes both 2D and 3D in the sense that the stacking is not only done in one vertical tower. An interposer or even a die can be used as the base to sustain multiple towers of stacked dies. This approach while not heavily adopted yet, can be used when some of the dies do not match in their physical characteristics to be part of a vertical stack, but still it is necessary to shorten as much as possible the interconnect. For example, in Figure 2.10 c, the physical dimensions of die 2 make it hard to stack with the other dies, or it could be that instead of size, its heat generation is considerable, so to properly dissipate the heat and not affect other dies, it should remain free of dies on top of it.

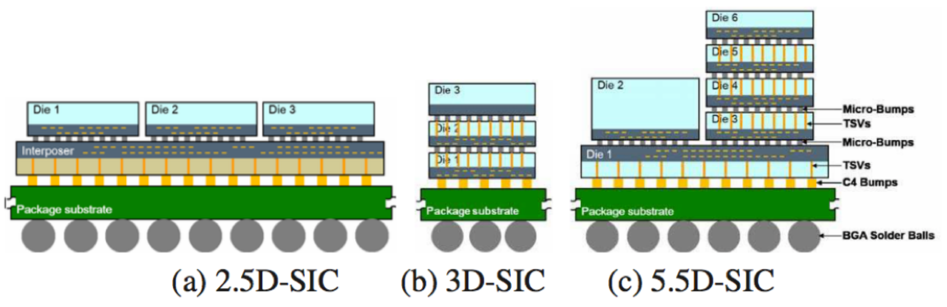


Figure 2.10: Stacking Architectures [34]

2.3. Challenges

As with any new technology, there are situations that need to be resolved. In the case of stacked ICs, it is possible to group the challenges into the following four areas.

- **Design:** Due to the new organization of the devices, it is required to develop new strategies with the goal of finding the best ways to establish the structure inside the dies while also considering the interaction between them. Floorplanning must be capable of exploiting to the maximum die area through optimal placement of functional blocks, while keeping an eye on the vertical location to ensure ways of taking advantage of the short interconnects brought by stacking instead of routing traces on a circuit board.

  As TSVs are added to the design, new rules regarding keep out areas must be introduced [9, 57]. Another important aspect to consider in the design stage is the location of areas that will experience high temperatures. These hot areas should not be placed close to each other and should also have an easy way to dissipate the heat, either by using TSVs as a way to extract the heat [58], or by being close to heat sinks.

- **Manufacturing:** On the manufacturing part of the process the main concerns are related to the new processing steps that are performed on the dies to prepare them for the stacking operations, as well as the stacking operations themselves. By having to apply these new steps to the dies, new defects may be introduced and also known defects may affect on a new way that did not occur before. Having to subject the dies to a more extreme thinning can damage the die's substrate affecting the transistors properties and weakening the device[13]. The stacking steps require a very exact alignment, to prevent TSVs and microbumps from not making good contact, thus increasing resistance or even leaving a broken connection [15]. Issues in the TSV fabrication, for instance voids in the filling of the vias, affect the resistance of the via[15].

- **Testing:** The stacked IC process affects greatly the testing approach to integrated circuits. On a 2D device, there are typically two test moments (opportunities to perform a test). The first one is the wafer test, which makes sure that the devices are working properly after the wafer processing and the second test moments is the final test, executed after dicing the wafer and packaging the dies; this test strives on finding defective devices before they get shipped to the customers, so for a 2D device, the test flow, which is the combination of tests performed at existing test moments, consists of a wafer test and a final test. For stacked ICs, the testing phase is more complex, spanning four moments instead of two. On the first moments, due to now there are more than one type of dies, the wafer testing must be performed to every kind of die that will be part of the stack, so for n dies, n wafer tests. In the 3D test flow, the

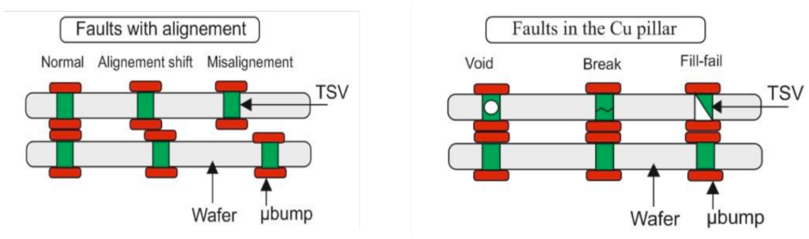


Figure 2.11: Common types of defects on stacking process [12]

wafer tests are called pre-bond tests, because as its name says it, they are performed before the dies are stacked. The next phase is the mid-bond testing. This stage consists of all the tests that occur after stacking the first two elements until the penultimate stacking operation. Following the *n* dies case, there will be *n - 1* stacking operations, so there will be *n - 2* mid-bond tests. Once the last stacking operation is performed, the post-bond test takes place. All the good stacks are then packaged and finally before being sent to the customers, they must pass one final test, for a total of *2n* test moments.

Not only the number of tests changes, but with the stacked case, what needs to be tested also will be different. New critical structures like TSVs are now part of the devices, so tests must be designed to find defects during their manufacturing and later on, after being stacked. As a product of this, test contents need to consider also this new class of defects.

Another challenge comes in the form of finding a test flow that has a very high coverage, while being cost effective. With the increasing number of performed tests, it is impossible from a cost point of view to always test with the highest coverage, so it becomes essential to find a strategy that maximizes the faulty dies detection, without incurring in unnecessary expenses.

Of similar relevance is the test access challenge. Typically tests are performed through probes on the wafer test and on the final test through the package's pins, but now partial stacks without access to the bottom die might exist and access to a particular die on the stack might be blocked by other dies, so new methods of accessing the dies for testing are necessary. One possible solution is the addition of probe pads on the backside of dies to grant access, but probe card technology for the required pitch is very limited [33]. A complement to this is to enable the dies with DfT structures that will help perform the tests and pass the data to a die externally accessible at that moment. IEEE standard P1838 is one possibility for this, seeking to implement boundary scan chains throughout the stack [34]. For stacks with memories, it is also possible to take advantage of them, to generate test patterns and apply them to the other dies utilizing the existing interconnect [53].

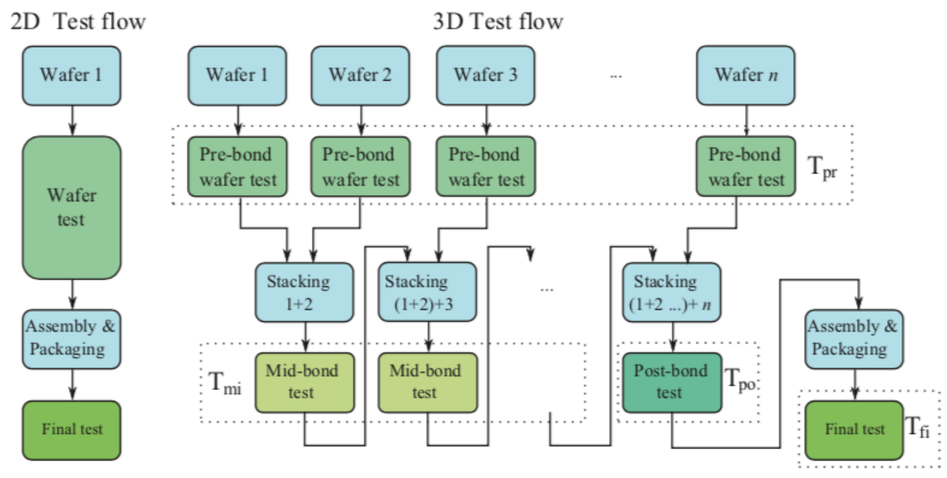


Figure 2.12: Typical test moments on a 2D and on a 3D flow [46]

- **Supply Chain:** Mostly, all dies will not be manufactured by the same company and it is even possible that the die will be manufactured by a certain company and TSV manufacturing or other parts of the process will be done by a different manufacturer specialized on that process. Based on that, a very large probability exists that wafers and naked dies will be transported between this different elements. This results in a different level of logistics coordination that needs to be considered, where a delay in one of the participants, can halt the whole stacking process. In contrast to only moving finished products, also important to contemplate are the risks that this entails in terms of physical damage to the devices.

2.4. Opportunities

The transition to a stacked IC model opens up possibilities in terms of gains to be achieved. As in the challenges, the opportunities are not solely focused on a specific point, but can influence different aspects of the stacked ICs.

- There are gains to be made related to cost. One scenario where this becomes evident is when what would have been a large die, gets broken down into smaller pieces. Smaller dies tend to have higher yield, so the stacks yield can be higher than the yield of the large dies [43]. By doing this, the number of good dies increases, because by being smaller, for the same defect density, less dies will be defective than in the case of larger dies [36]. Also related to cost, by reducing the footprint of the ICs, savings can be made related to materials used and the whole device were the stacked IC is being used can be of a smaller size.

- The flexibility introduced with the stacked IC technology, enables mixing and matching of dies with different functions into one package. This flexibility is extended into the manufacturing process of those dies, where non critical functions can remain on older processes with higher yields and only the most advanced functional blocks will be manufactured on a more advanced process. This in contrast of a 2D die, where the whole die can only be manufactured under one process and this being set by the chips most complex functions. A byproduct of this is that time to market can be greatly reduced, because instead of having to design the whole IC, existing and proven parts can be used with a few adjustments and only the novel elements have to be designed from scratch.

- With the shorter interconnects achievable through stacking, significant benefits can be attained regarding the electrical performance of the ICs. Reductions in wire length lead to less power dissipated and employed to boost signals to travel from one location to another[17, 38]. Also the delays are shorter, improving signal propagation times and clock distribution. There is less chance for signals to be affected by noise by having to traverse shorter paths, reducing the possibility of being affected by other elements.

- From a reliability point of view, new possibilities arise to extend the lifetime of the chips. In the memory world there are techniques to repair memory elements, spare memory elements are included on the chips, to allow the chip to not lose its capacity when a few sections break up. When this occurs, the defective elements are not used anymore and spare elements replace them through internal routing[25]. With a stacked IC it is possible to extend the number of spare elements, so if there are more than one memory dies on the stack, it could be possible to share between them the spare elements, prolonging the life even in cases when all the local spares are already in use. Similar to the memory case, for other kinds of chips, it could be possible to move part of its tasks to another die, that also has that functionality.

- From a security point of view, there are also benefits, even though this is not one of the reasons that would motivate the transition to a stacked architecture. By stacking, it becomes much more difficult to reverse engineer an IC than a regular 2D, because the processes used are very invasive and destructive. It would be necessary to separate the layers, which has a very high propensity to damage the devices. If an approach based on imaging technologies is used, it will be very difficult to gain a good understanding of what is going on, because from all the dies will be overlapping on top of each other in the images.

3 Cost Modeling

Decision making is an essential skill in the pursuit of achieving goals. It ranges from the most basic aspects of our lives, to complex situations as a company deciding on developing a new product line and finding the best way for accomplishing it. Of great importance for an effective decision making is access to accurate and useful information. Having this data will shed a light on to what is going on in the process in question, furthermore, it will help understand how different factors affect the process to come up with a strategy that maximizes the benefits. Different methods exist for aiding on this important task and cost modeling is one of them.

As defined in [41], "A Cost Model is a mathematical model that needs parametric equations for estimating the costs of a product". In other words, a Cost Model can also be seen as a representation of a process through the use of equations that describe what is occurring, and as the output, it provides an estimate of the cost incurred on that process, based on the input data and the equations.

Cost Models are employed in many areas, but for this work, we are going to focus on the application of Cost Models to stacked IC development. Section 3.1 will introduce a classification for the state of the art. Section 3.2 describes 3D COSTAR and Section 3.3 highlights the limitations of 3D COSTAR.

3.1. Classification

By examining the literature, it is possible to find different instances of cost models applied to the stacked IC development. For each model there is a set of equations that describe the inherent process. Although these equations vary due to methodology and differences on the focus the authors had when developing their studies, there are valuable contributions that can be extracted from them in terms of ideas and methodologies.

Figure 3.1 shows the way the studies were classified. Due to the large differences in the manufacturing process the first categorization is made based on the three 3D technologies, the reason for this is that although all produce 3D SICs, the process involved on producing each technology largely differs from each other, so it is logical to make a separation based on this. It is not the same to stack individually packaged ICs, than to directly stack the active silicon layers on top of each other to obtain a monolithic 3D SIC.

After the technology classification, the studies are displayed, mentioning the author's name and year of publication, based on the cost classes that are analyzed on the publication. The only different case is for 3D COSTAR which will be later explored in depth. There are 5 cost classes onto which the supply chain for the process of producing a stacked IC has been separated. These classes are:

1. **Design:** Covers the activities related to the design of the SIC.
2. **Manufacturing:** Contains all the steps taken during the actual manufacturing of the SICs. It includes the manufacturing of the individual dies as well as the stacking process, up until the complete stack is achieved.
3. **Testing:** In charge of preventing defective dies to pass as good and be sent to customers. As mentioned before, instead of the 2 test moments for a 2D IC, stacked ICs can have up to *2 x number of dies* test moments, so it is necessary to consider all those possibilities.
4. **Packaging:** Describes the process that takes place after the final stacking operation, where good stacks are packaged, to provide protection to the dies and the means to communicate with other elements.

5. **Logistics:** Because of the large possibility that not all dies are manufactured at the same place, it is expected to transport them throughout the process. Another factor is that not every manufacturer is capable of performing all the steps for stacking, so the logistics class keeps track of these movements, because they also contribute to the cost.

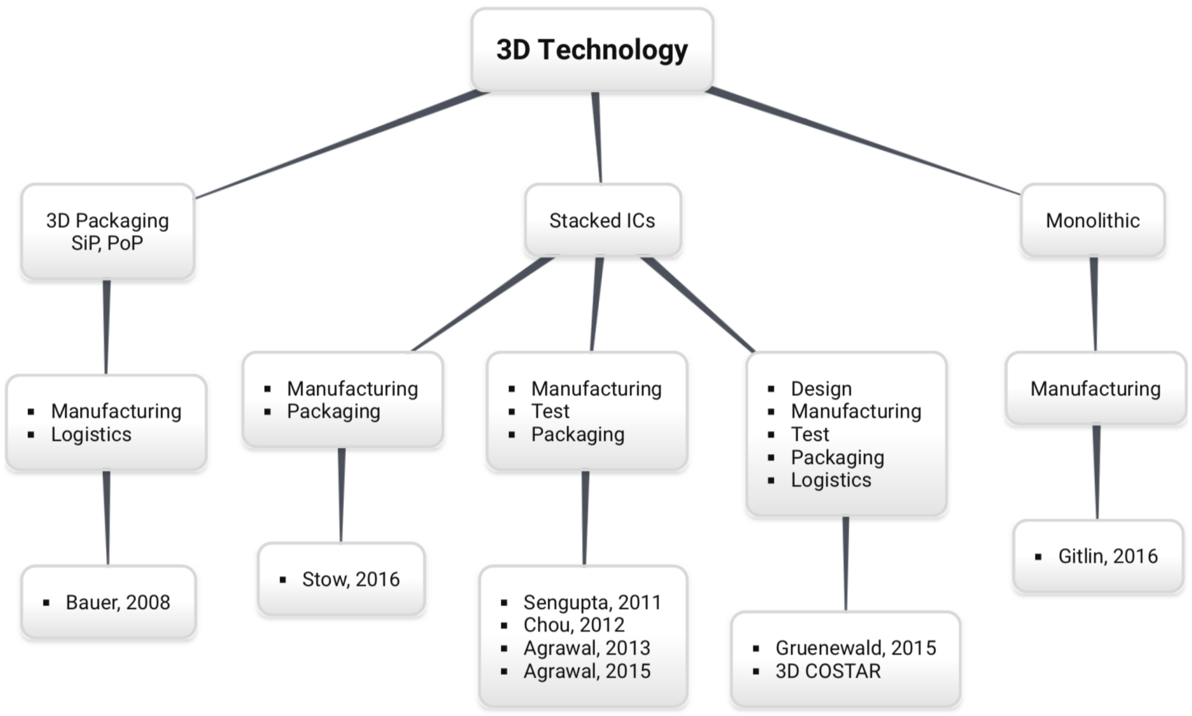


Figure 3.1: Existing Cost Models Classification

The first study seen in Figure 3.1, corresponds to the work done by Bauer on [10]. This work is focused on the **3D Packaging Technology**, including both System-in-Package and Package-on-Package approaches. The main objectives are to evaluate the opportunity cost of manufacturing SiP and PoP and also finding out how logistics, inventory and time to market impact the cost. The study uses a very simple cost model, with few variables and basic calculations performed through an excel flow to obtain the results. Another drawback is that although it includes some aspects related to logistics, it is mainly concerned with a limited manufacturing model description and it is not suitable for 2.5D and 5.5D cases.

Moving into the **Stacked IC technology**, there are three main groups to which the cost models fall into. The first group contains the manufacturing and packaging classes. In this group, the cost model presented by Stow and its collaborators in 2016 [44] is found. The main idea behind it is to present a cost model for 2D, 2.5D and 3D based on fabrication cost, while also including a thermal model for heat sink calculation and packaging costs. The approach taken by the authors is to create a model that focuses deeply on the manufacturing process on a very micro level, as can be seen in the equations, where details as wire length are used to find number of metal layers. A similar approach is used on the packaging process, where deep physical properties are analyzed to find the thermal characteristics of the device. Although very detailed on the classes that it explores, this model leaves out many more than half of the classes needed to give a more complete cost picture of the stacked IC production process. Furthermore, for the elements that it does include, it just provides the results for the input data. It does not perform any kind of analysis that could help reduce the cost, by optimizing parts of the process.

The next group belonging to the **Stacked IC Technology** is formed by models that besides manufacturing and packaging, also consider the testing phase of the process. In this group there are 4 different cost models, each with its unique approach to the task.

The model proposed by Sengupta in 2011 [42], seeks to find the best test flow, based on two characteristics, test time and number of input dies. While the model itself does not explore deeply the manufacturing and packaging processes, it does evaluate different possibilities regarding testing. The methodology utilized is based on creating 3 different testing scenarios, where different test strategies are followed. The first scenario performs the pre-bond test to all dies and for the mid-bond and final test only the interconnect is tested. The second scenario only performs die and interconnect test once the stack is complete and on final test. The third possibility tests at all possible test moments. While there are some issues with this model as the few parameters to build the cases or that it assumes a 100% FC, it is important to highlight the analysis performed to the testing phase. By doing this, they were able to incorporate to the model the capacity to analyze different possibilities and provide a local optimum for the problem in question.

Still addressing the same three classes, the next model on the classification is the one developed by Chou in 2012 [16]. The aim of this study is to develop a cost model that includes manufacturing and testing for interposer based stacked ICs, manufactured through Die-to-Wafer or Die-to-Die processes. Besides the model, a tool that employs the model is created. This tool is capable to perform an analysis on the test flow, by iterating through the process while modifying the parameters related to test, allowing to make comparisons of the results obtained. On the paper, 6 test moments are identified, by performing tests with different coverages or not performing them at all, the different test flows are constructed. As in the case of [42], by including the test phase, more possibilities come into play when calculating the cost,Chou identifies this and ads it to his tool but on a limited manner, by having to manually generate the different possibilities. By focusing only on interposer solutions, they are able to relate on a very detailed way the manufacturing and testing processes, but it also makes it fall short in covering more cases from the **Stacked IC Technology**.

The last two models from this group were proposed by Agrawal and Chakrabarty. The main objective for both cases is to find a test flow that minimizes total cost, while employing a heuristic to guide the search of that particular case. The main difference between the two is the heuristic used, which also affects part of the equations that make up the cost model. On the first model [5], the optimization method is based on a matrix partitioning solution. It works by creating a matrix conformed by the yields of the dies and of the stacking operations. The matrix is then partitioned based on the test moments executed, this partitions cause the grouping of the yields from the beginning of the partition (one test moment) up until the next partition (next test moment) allowing them to analyze the impact of different test flows, thus translating the test flow selection problem into a matrix partitioning one.

The second model by Agrawal and Chakrabarty [6] and last belonging to the current group, modifies the test flow optimization by changing the method to using A* to guide the search. The manufacturing cost model is slightly modified to improve the cost calculation, but in general terms, the difference between the two studies lies in the optimization methodology. The idea behind this paper is to formulate the optimization problem as a search problem and solve it by employing the A* algorithm. The solution is based on creating a decision tree, where the different states represent the application of a test with a specific coverage. At the end of the process, they are capable of providing a test flow that minimizes the cost for a scenario based on manufacturing, testing and packaging. While very thorough regarding the included classes, these studies falter in the sense that the scope does not include other important factors that affect the stacked IC cost.

Following in the classification comes the group of cost models that include all 5 classes. In this group 2 cost models exist. In this section only the one by Gruenewald [22] will be explored, as 3D COSTAR will be discussed later on the next section.

Besides a cost model which includes all 5 major classes, Gruenewald and his group also developed a software tool used to perform the calculations. As in all previous cases, the model formulation has its own distinct approach and the fact that all phases are part of it, is noteworthy. As a shortcoming, it can be mentioned that the tool only has the capability to calculate the cost for one case. Although it does provide results based on the different classes, it is cumbersome to compare test flows, as there are no ways to specify other cases different than rewriting the case to analyze.

Moving on to the **Monolithic** cost models, only one was found. This model belongs to Gitlin [21] and it looks to model the cost for a monolithic process in terms of complexity, yield and area. This model only focuses on a very small part of the whole process, using a basic approach due in part to the experimental aspect of the technology which it belongs to. It uses very few parameters and as the output, it relates the cost of manufacturing a specific device on a monolithic 3D process in comparison to what it would have taken on a 2D process.

After exploring these cost models, it can be said that an option that provides the whole picture does not exist. There are models that perform some analysis to find the best test flow in terms of cost or other models that include all five classes, but there is no solution that does it all. Therefore we consider that the stacked IC field is incomplete with regards to cost modeling tools to help plan and execute the production process.

3.2. 3D COSTAR

Proposed by Taouil [47–49, 52] 3D COSTAR is the tool based on a cost model that encompasses all 5 classes of the stacked IC production process. Its main goal is to calculate cost of SICs, while considering the whole supply chain (design, manufacturing, ...) in order to provide: The overall cost as well as the cost breakdown, product quality in DPPM (Defective Parts Per Million) and a sensitivity analysis, which runs similar cases with slight modifications in selected input parameters to observe the effect in the overall and each class' cost.

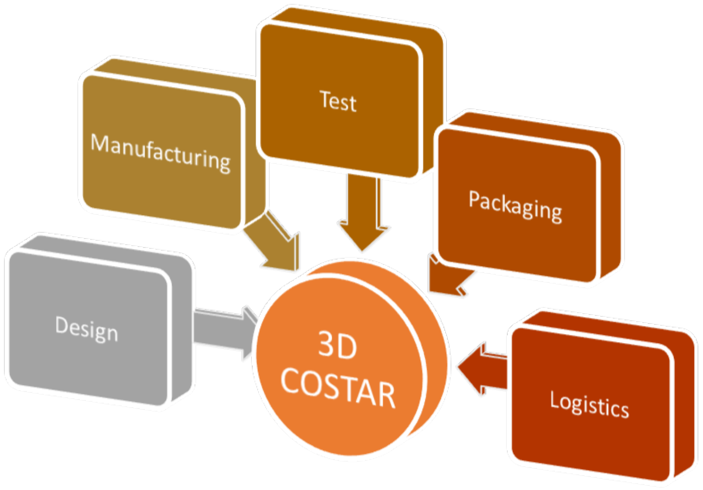


Figure 3.2: Cost Classes

As discussed previously, 3D COSTAR considers all five cost classes, Figure 3.2 shows the classes involved, where each class has its own group of parameters that are used to estimate the cost.

Design

Besides the typical design responsibilities found on a 2D IC process, stacked IC manufacturing makes it essential to consider the effects of the different elements related to the 3D aspect of the design. Either taking into account factors as the required area to accommodate the TSVs with their keepouts to avoid physically damaging parts of the die or inducing noise, to having a plan with respect which dies will be accessible for testing during intermediate steps of the stacking process, and ensuring the existence of an I/O structure to perform those tests are examples of situations that affect the cost of a stacked IC.

Manufacturing

This class is in charge of modeling all the processes related to the manufacturing process. Due to the nature of the manufacturing process and the delicacy of the materials used, not all dies will survive this stage, with some breaking during the die manufacturing and others during the stacking process. Primarily, this class will be comprised of the yield and the cost for each manufacturing operation. Due to the clear separation between what constitutes the wafer processing and the stacking process, the manufacturing class is subdivided into those two elements.

The wafer processing subdivision includes the die manufacturing yields as well as the interconnect (TSV) fabrication yields, internally in the tool, the yields are unified under a general manufacturing yield, which is the product of the die and TSV manufacturing yields. The related costs are also part of this subclass. Each manufactured die entails an increase in the cost. Defective devices, either due to the die manufacturing or

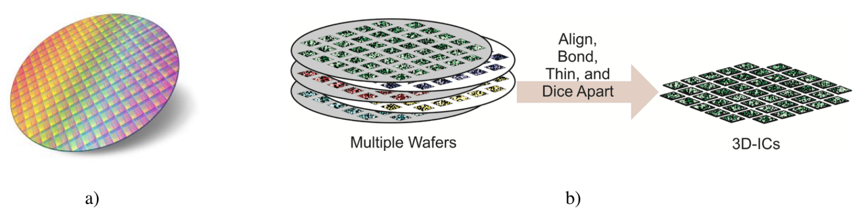


Figure 3.3: a) Wafer processing [56]. b) Stacking process [54]

the TSV creation, are inevitable. This is a consequence of the manufacturing process, which is not perfect. Those defective devices are considered in the cost by having to fabricate more devices of each die than the final number that will be shipped to customers. The model is also capable to calculate the yield with the negative binomial formula[14].

The second subdivision belonging to the manufacturing class is concerned with the stacking process. In it are located the yields and costs for each stacking operation. As in the wafer processing case, in the stacking process, die yields and interconnect yields are also considered separately during the parameters input, but internally are unified through the product of both yields. Similarly, the number of times a stacking operation is performed is counted to obtain the stacking cost, no matter if the resulting stack is defective or not.

Testing

The testing class is of great importance as it will filter the defective dies and stacks from the good ones, thus minimizing the possibility of shipping defective products to customers. At different manufacturing stages, defects can occur to the dies, thus the necessity for performing tests. Different tests can be performed to the dies as well as the interconnect, in some cases targeting a particular die or more than one member of the stack. These tests are also separated into die related or interconnect related, having for both cases cost and fault coverage parameters. In terms of test moments, this means pre-bond, mid and post bond and also final test after packaging. It is possible that defective dies pass the test, this elements are called test escapes and are also tracked by the model, because they form the base of the quality measurements.

The test cost is calculated by accounting the cost for each test, based on the number of times it is performed. As mentioned in Section 2.3 under the testing challenge, on a SIC process, there are many test moments, which increase with the number of dies in the stack and it is not mandatory to perform all tests at each moment. If not properly planned, it is probable that unnecessary or not enough tests will be performed, thus increasing the cost. Another factor is the fault coverage tied to the performed tests, which has repercussions in the cost, due in part to the cost of performing more thorough tests, which are more expensive. Also related to the fault coverage is by having it being low, many defective dies will not be detected and sent to the customers, which will later return them, incurring in more costs due to having to replace the defective units and damaging the image of the company.

Packaging

After all stacking operations, remaining stacks go through one final manufacturing step where the tack gets encapsulated to prevent damages and also the external interconnect is provided. Due to the physical nature of this step, it is possible that defects get introduced into, up to that point, good stacks. To take into consideration this class, it is modeled in 3D COSTAR through a die packaging yield and an interconnect yield. Those yields characterize how the packaging process can introduce defects into specific dies or interconnects present in the stack. Besides the yields, which are used to keep track of how many good stacks remain in the end, there is also a cost for performing this part of the process, which contributes to the overall cost, based on the number of stacks that go through the packaging process.

Logistics

On 3D COSTAR five companies/houses are identified to be involved in the production chain of an SIC. Based on that, many different possible scenarios exist regarding the movement of dies/stacks to complete the process. It is possible that the whole process, since the design phase until packaging is done by only one company at one location, but it is also possible that the wafer manufacturer does not possess the capabilities to manufacture the TSVs, or the testing equipment, so they would require another entity to take care of this. Every

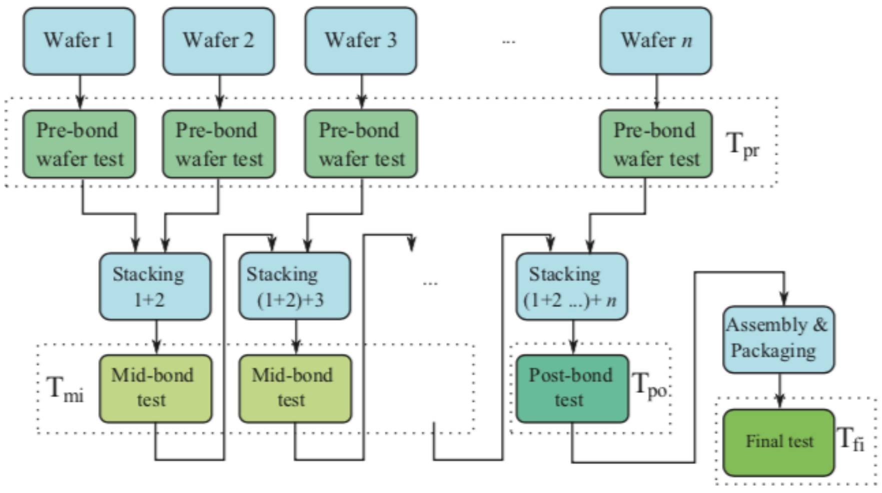


Figure 3.4: Test flow for a SIC [46]

move between one house to another carries a cost. In 3D COSTAR, 12 possible costs identified by letters as seen in Figure 3.5 are considered. These letters represent possible movements, this way, the tool is capable of taking into consideration the costs incurred.

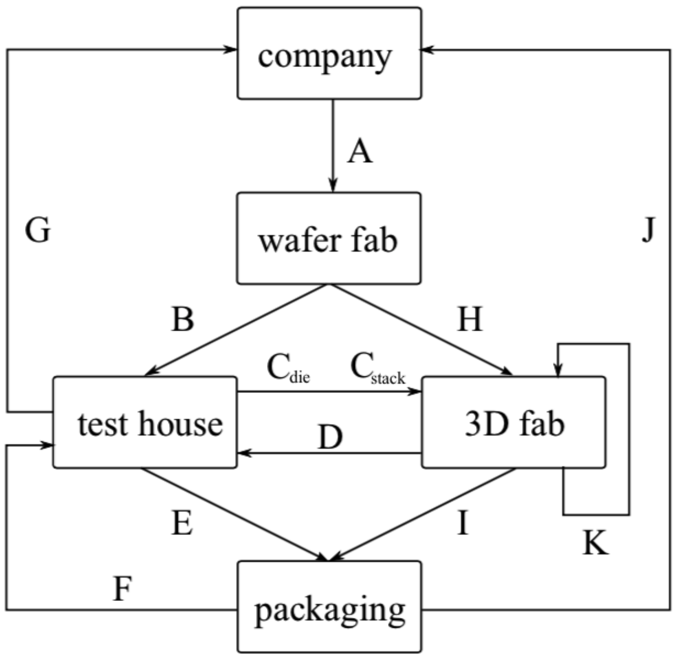


Figure 3.5: Logistics costs [52]

- **A** Cost from *design company* and *wafer fab*
- **B** Cost from *wafer fab* to *test house* to perform pre-bond test
- **C**$_{die}$ Cost from *test house* to *3d fab* for the dies that pass the pre-bond test and will be stacked
- **C**$_{stack}$ Cost from *test house* to *3d fab* for partial stacks that pass a mid-bond test and will have more dies added
- **D** Cost from *3d fab* to *test house* for stacks after being manufactured and still need to be tested

- **E** Cost from *test house* to *packaging fab* after post-bond, for the packaging process
- **F** Cost from *packaging fab* to *test house* to perform the final test
- **G** Cost from *test house* to *design company* after final test
- **H** Cost from *wafer fab* to *3d fab* for stacking without pre-bond test or for 3D structures fabrication
- **I** Cost from *3d fab* to *packaging* for packaging without post-bond test
- **J** Cost from *packaging fab* to *design company* when no final test is applied
- **K** Cost from moving stacks between 2 different *3d fabs* or different locations of the same *3d fab*

3.2.1. Tool flow

The way 3D COSTAR functions can be seen in Figure 3.6. The way the tool communicates with the user is through text files. The first step is to read the input file to obtain the parameters to be used in the model's calculation. At this step, the read values are verified to ensure that the data is correct. This means checking whether correct number of values are assigned to existing dies and interconnects.

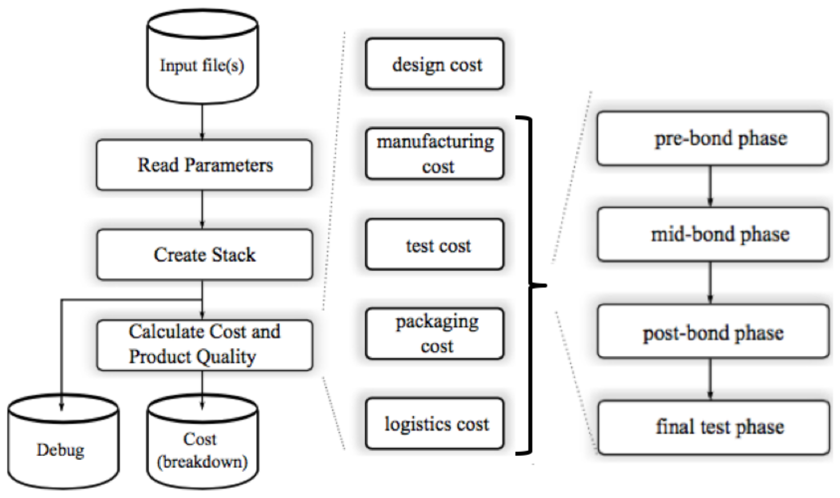


Figure 3.6: 3D COSTAR flow [48]

Once the parameters are verified, the stacking process based on the input values is followed to create an internal representation of the stack to verify that a valid stack is being created. This process is detailed in Figure 3.7. In (a) the final stack is shown. It is comprised of five dies, thus four stacking operations, where die 1 is the common base for two separate towers; the one formed by dies 2 and 5 and the other one formed by dies 3 and 4. Part (b) of the same figure demonstrates how this structure would be specified in the input file. The first stacking operation OP1 is described through stack_id1, where die 2 is stacked on top of die 1 in a face to face (F2F) orientation through a D2W type of stacking. The second operation OP2 consists of die 4 on top of die 3. This operation is back to face (B2F) as the back of the base die is in contact with the face of the top die and since both dies are not on a wafer anymore, D2D is used. By the end of OP2, two partial stacks exist, the one containing dies 1 and 2 and the other one with dies 3 and 4. The next step (OP3) joins the two partial stacks into one, by stacking die 3 (which had die 4 already stacked to it) on top of die 1. The final operation consists of stacking die 5 through its back on die 2's back to obtain the complete stack. Part (c) of Figure 3.7 shows on a basic level, how the stacking process is followed and stored inside the tool. An index is assigned to each particular operation (stack id: i) to identify it. For each operation, a list of the dies present on the stack is kept ($L_{D,i}$), as well as the interconnects ($L_{I,i}$), in addition to other related statistics as the dies on the top and bottom of the stack.

After creating and verifying the stack, all the required elements are in the tool. The next step is to employ the model to calculate the cost and the breakdown. It is at this moment that the parameters related to each of the five classes are used. Throughout this process, statistics are being recorded which are used to complement

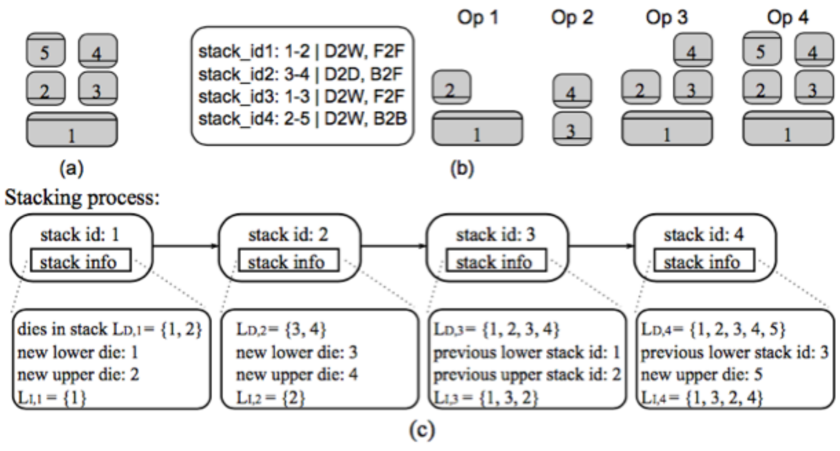


Figure 3.7: Stack representation in 3D COSTAR [51]

the main output where the actual cost and breakdown are reported. Those complementary elements can be seen on the debug file, which not only reports the final results, but also revisits the input data to confirm that the proper values were used, as well as partial results obtained from the intermediate steps.

3.2.2. Sensitivity Analysis

A differentiating factor between 3D COSTAR and the other models is the presence of the sensitivity analysis in 3D COSTAR. The way it is implemented is that the user needs to specify which variables it wants to explore. For those variables a sweep will be performed based on a range guided by a step size, both indicated by the user on the input file. For the initial value, steps and end value of the range, the cost calculation is executed and the results for all the cases reported through a Sensitivity summary file, additionally, the final results for each execution of the cost calculation and debug files are also reported back. Based on this, the tool is able to automate the generation of new cases based on the the original one and easily compare different scenarios based on the swept elements.

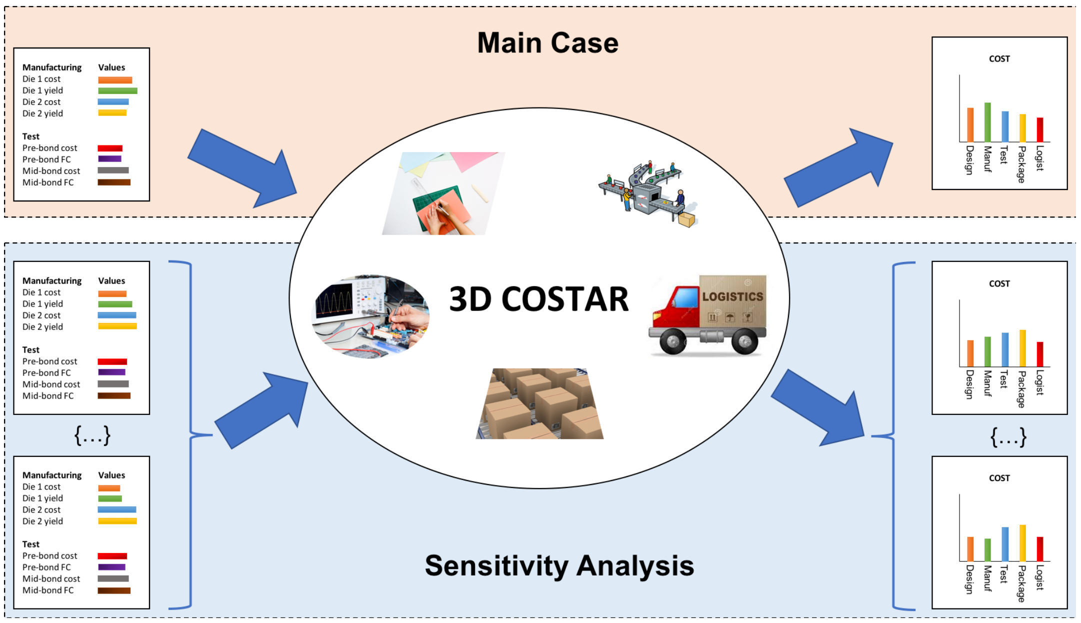


Figure 3.8: Main case and sensitivity analysis

3.3. Limitations

From the previous sections, we can say that 3D COSTAR is the most complete cost model and tool for the stacked IC process. It covers all five classes of the supply chain while adding extra value through the sensitivity analysis, introducing a level of automation in an effort to examine more cases with the hope of approximating to a best case solution. Nevertheless, there are still some characteristics of the tool that have room for improvement which will be explained below.

- **2D mode is not present** While the model is capable of working on a 2D mode, the way the tool is constructed, it is impossible to run a real 2D case. In the occasion that the user wants to simulate this scenario, it is possible to approximate it through the use of dummy elements with zero cost and a 100% yield, but this approach requires tricking the tool and is mistake prone.

- **No automatic way of comparing different stacking strategies** The tool creates a representation of the stack based on the specified stacking operations. It is possible that while keeping the same stack structure, altering the order of the stacking operations will reduce the cost, by in some cases shifting particular operations either earlier or later in the process . This could potentially lower the cost by for example not having to subject expensive stacks to a low yield operation at the end of the process, which could potentially decimate the number of good stacks gravely.

  Augmenting the tool with the capability of reordering the stacking operation's order of execution and performing the cost calculation for all the different orders could be useful for the user, as it could suggest a different order from the one he had in mind that could lower the cost. Not all cases will behave equally, so it is not possible for designers to assume based on a general rule, therefore a way of simulating this is beneficial.

- **Inaccessible dies during specific test moments** Not all dies included on a stack have the required structures to enable I/O access for testing purposes. To administer an external test, landing pads must be provided for the probes or micro bumps capable of resisting the effects of repetitive use and not all dies include them [33]. The tool does not consider these scenarios, all test moments are valid, while in reality there are test moments where this is not the case.

- **Lack of differentiation between die and TSV costs** On the manufacturing class, the cost does not differentiate between the cost of creating the logic on the die and that of the TSV structures, it is one general parameter. As mentioned during the logistics breakdown, it is possible that a die manufacturer does not posses the capability to create the TSVs, so another company will have to do the TSVs. In terms of results analysis, having separate manufacturing costs enables for a deeper cost breakdown, enabling the analyst to better determine which parts of the process have a greater impact on the product's cost.

- **No optimal test flow determination** The tool only considers the case laid down in the input file. Similar to the stacking order, analyzing different test flows can provide significant gains with respect to the cost.

  Determining the correct strategy regarding fault coverage and test moment for a 2D case requires a non trivial effort. Coupled with the fact that for a SIC the number of possibilities increases significantly, optimal test flow determination becomes a very complex problem. Delving into this, in the SIC scenario besides the traditional logic tests, there is an additional component to testing, which is the interconnect test. At every possible test moment, it is now necessary to also consider this new component in the determination of the tests that will be applied with their respective fault coverages and costs.

  Another aggravating factor is the relation between the number of dies and the number of tests moments, which is described as $2n$ test moments for n dies. As stacks become more complex, with more dies in them, the complexity of determining the best approach increases substantially.

  Introducing a way to evaluate different approaches to testing, has the potential of enhancing the usefulness of 3D COSTAR greatly, as cracking the optimal test flow problem offers significant benefits regarding lowering the cost of the SIC production process.

4
Extensions to 3D COSTAR

The main goal of this project is to extend 3D COSTAR with new functionality. These new functions must solve an existing limitation found in the tool, therefore providing an improved and more complete solution for the users. This chapter explores the proposed extensions to 3D COSTAR, highlighting how they are able to solve the related limitations, as well as a high level description of their functioning.

Section 4.1 discusses the test flow optimization, proposing a solution through the shortest path problem and exploring different algorithms to find the best candidate. In Section 4.2, the limitation associated with the stacking strategies is addressed, by rearranging the stacking operations in all possible orders and calculating their costs. Finally, Section 4.3 will explore the addition of a new cost to the manufacturing class, which will specify and keep track of the cost of manufacturing the TSVs.

4.1. Test Flow Optimization

One of the strong advantages of 3D COSTAR is its detailed approach to the test class. It covers the whole manufacturing process, with tests for each die and die and interconnect tests. In terms of moments, it includes from the pre-bond phase, up to the final test after packaging, going through mid-bond tests, available after every stacking operation, where individual dies or interconnects can be targeted as well as testing the partial stack. All the test with different possibilities in terms of fault coverage and cost. Based on this and as mentioned before, for n dies, there are $2n$ test moments and for each test moment, the number of possibilities related to which kind of defects are targeted and how thorough the tests will be performed, can become immense. Defining a test flow is a multi-step process. For each test moment, first is necessary to define if a test will be performed or not. If it will be performed, then the problem transforms into establishing what will be tested or the targets of the test. The possibilities range from only one die or interconnect, to testing the complete stack at that particular moment. After selecting the targets, another decision must be made which pertains to the quality of the tests that will be performed. All these, while also considering how one test moment relates to the others, so as to maximize the quality at the lowest possible cost. With all this information, 3D COSTAR is able to provide a very detailed cost breakdown for the specified test flow, but it does not offer any kind of assistance in case the user wants to find the cheapest scenario from a particular set of options. These options can be other possibilities that were discarded for the original case, but that the user would still like to explore and compare due to the possibility of finding a cheaper scenario. For example, for the original case a specific test was chosen, but the test engineer might also have five other tests with distinct fault coverages and prices at hand, so it could be valuable to recalculate the cost with a combination of those tests and see if a better case exists.

The complexity of defining a test flow comes from the fact that it is not straightforward to predict the impact of a test. For a particular case, it could be thought that a more expensive pre-bond test, with retesting of the dies during the stacking operations will increase the total cost. It could be assumed that there is no need of a high quality pre-bond test because defective dies will eventually be detected later in the process. However, by better filtering defective dies before they become part of the stack, fewer stacks will have to be scraped later in the process, thus reducing the waste of materials and time due to having bad dies advancing through the process, so this scenario could make the overall cost be lower in comparison to the same scenario but with a lower coverage at the pre-bond tests.

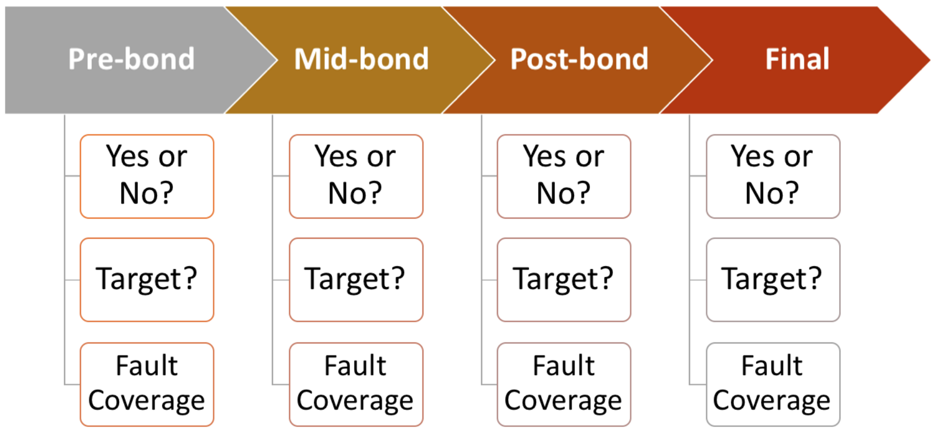


Figure 4.1: Conformation of a Test Flow

This test flow optimization problem can be easily and reliably described as a shortest path finding problem. To map the different possible test flows, a tree structure is employed, where the root node is the original case from the main file and the particular tests that are being examined are mapped into the branches. The first test moment that will be evaluated corresponds to the children of the root node. Here, each possibility related to the first test moment will be represented by a particular node. Furthermore, if there are more test moments that need to be inspected, they will continue branching out from the previous node, with this branch repeating itself in all previous level's nodes.

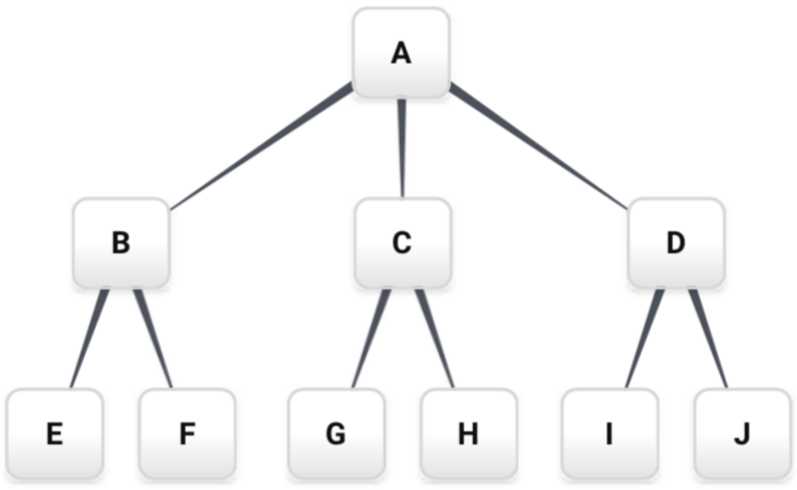


Figure 4.2: Depiction of test flow problem through a tree

An example of how this problem will be represented as a tree is given in Figure 4.2. To relate this scenario to a **test flow optimization** problem, a simple case is proposed:

The stack is formed by three dies $d = \{1,2,3\}$ and two stacking operations $st\_ops = \{1,2\}$. In the first stacking operation, die 2 is stacked on top of die 1, thus the other operation will stack the remaining die on top of die 2. As for the original test flow, pre-bond test (wafer tests) to all three dies are applied, with a fault coverage (fc) of 94% and a cost of \$0.50 per test. After each operation is performed, the dies involved are tested with fc = 96% and a cost of \$0.75, so for $st\_ops = 1$, $d = \{1,2\}$ will be tested and after the other operation, all three dies will be tested. Once the stacks go through the packaging process, a final test is performed to all members of the stack before being shipped to the customers, this time with fc = 99% and a cost of \$1.

Based on other possible tests available to the user, he would also like to explore the impact on the cost of this process if he modified two of the test moments previously stated. The first case will be to modify the test applied to die 1 after the first stacking operation. For this test, three options will be presented, the first one with fc = 95% and cost \$0.65, the second fc = 98% and cost \$0.90 and the last one fc = 99% and cost \$1. This three options are represented in Figure 4.2 with the nodes B,C and D respectively. The way to interpret it is that node B is similar to the original case, but with fc = 95% and cost \$0.65 instead of the original values for that particular case and so on for nodes C and D. The second case is for the final test of die 3, where two new possibilities exist, the first one is fc = 95% and cost \$0.65 and the second one fc = 97% and cost \$0.85. These options are depicted in the tree at the lowest level, with the left branches (E, G, I) always representing the first possibility of case two, but each for a different first case value, as can be seen by following the branch to the parent, so node E is a test flow where the original case has been modified with option one for case one and option two for case two; node G is option two from case one and option one from case two. The right branches (F, H, J) representing the section option for case two, also each one related to a different scenario from case one. It is very important to note that the complete and valid test flows are located in the lowest row of the tree. Nodes E, F, G, H, I, J are the only ones that contain complete test flows. All nodes in intermediate levels, B, C, D in this case, only contain partial test flows, in those nodes, no decision has been made regarding the test moments evaluated at the next level, thus they possess incomplete information regarding the test flow.

The following sections will discuss the shortest path problem and possible solutions, with emphasis on the formulation of the algorithms in response do the **test flow optimization** problem. After that, the actual solution employed in the tool will be explained.

4.1.1. Shortest Path Problem

Belonging to the graph theory branch of mathematics and more specifically to the route problems domain, the shortest path problem is in essence, a problem in which the solution is achieved by finding the shortest path(lowest cost) between two nodes in a graph, where there is a cost associated to the links between nodes. Applications for this approach are large in number, ranging from the easily relatable case of finding the shortest path on a map, implemented in navigation systems, to any kind of problem that can be represented through a graph, with the nodes representing states and edges the transitions, shortest path can be used to find from the possibilities, the one that requires the least effort. Besides the previous example there are many other fields that employ this approach to find solutions, for example, in telecommunications it is used to find the connections with the shortest delay and other metrics that allow the functioning of routing protocols, in production lines to analyze the balance between inspection cost and production cost, it is also used in robot navigation systems and video games for non-playable characters movements, etc[7].

Different ways for classifying available algorithms exist, for this case, a classification based on the strategy each algorithms follows to traverse the graph. This classification can be seen in Figure 4.3.

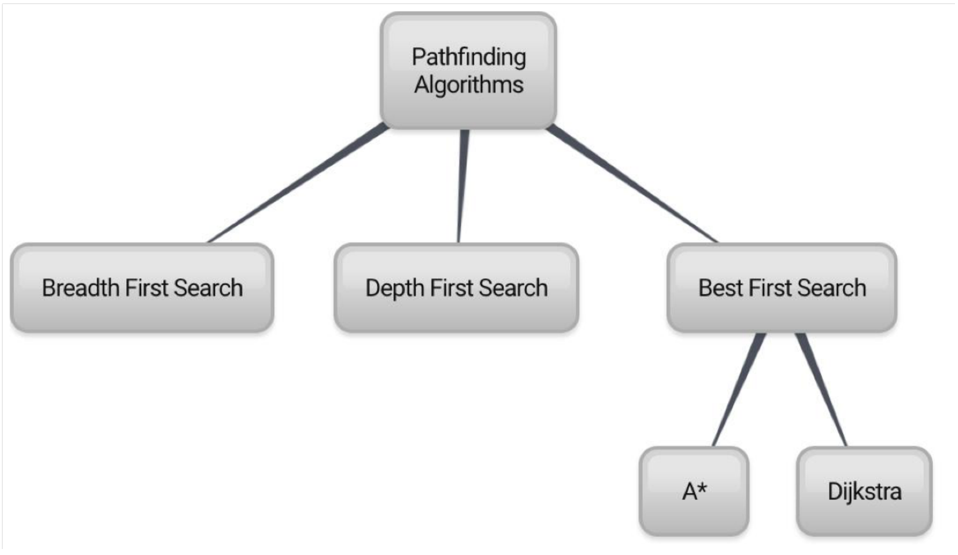
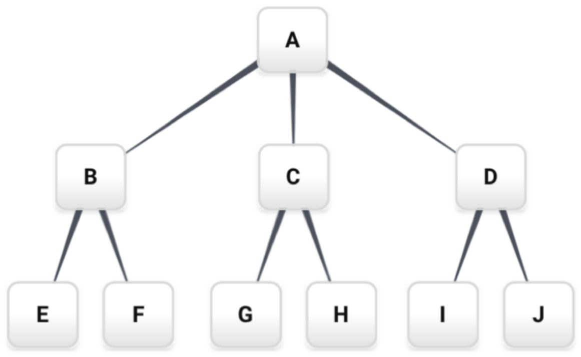


Figure 4.3: Shortest Path Classification

The following sections will explain these algorithms with the assistance of an example to more clearly illustrate how they operate. The example case depicted in Figure 4.4 consists on a three level tree structure, where the root (node A), has three children and each of them has two children, this last nodes are commonly known as leaves. Each path represented via a black line has a value attached to it, which specifies the distance between the points joined by the line. The total space consists of ten nodes, with three of them at an intermediate level and six leaves. Next to the tree, a table lists the respective distances between the pairs of nodes. The distances are known only after the path is explored, this means, that when moving from A to B, the distance for that path is discovered and updated at the destination. Although this is a very small example, it will be constructed in a way that will make it possible to visualize how each of the algorithms differs from the others, thus allowing to choose the most favorable option to implement in 3D COSTAR with the goal of optimizing the test flow formulation.



Path	Distance
A-B	7
A-C	5
A-D	2
B-E	4
B-F	6
C-G	7
C-H	9
D-I	4
D-J	8

Figure 4.4: Tree Structure used in Examples

By following an exhaustive search approach, where all the possibilities are calculated, the total distances traveled between node A and each of the different end nodes are shown in Table 4.1 from where it can be seen that the path that goes from node *A* to node *I*, through node *D* is the shortest path with a total distance of 6 units. With these results, it will be possible to compare the algorithms based on how they found the shortest path and from there select the one that will work better with the 3D COSTAR case.

Table 4.1: Total distances for example case

Path	Total distance
ABE	11
ABF	13
ACG	12
ACH	14
ADI	6
ADJ	10

Breadth First Search

The idea behind this algorithm is to guide the analysis by visiting all the neighbors of a node before moving on into the next node. In a tree structure, it means from the starting node, visiting all of its children, once a node is visited, it is placed on a list or queue following a FIFO (first in, first out) policy, based on the order of visit. Once all of the children of current node have been visited, the first element that was added to the list becomes the next current node, where the same process of visiting its children is performed. This is repeated until the whole tree is explored.

Figure 4.5 demonstrates how the example's tree structure would be analyzed by using the **breadth first search**. The numbers next to the arrows represent the order in which the nodes are visited, with the colors being used to mark all the paths that correspond to one parent node, thus will be visited when exploring that element. Following the rules guiding this algorithm, the first step will be to visit all of root node *A* children,

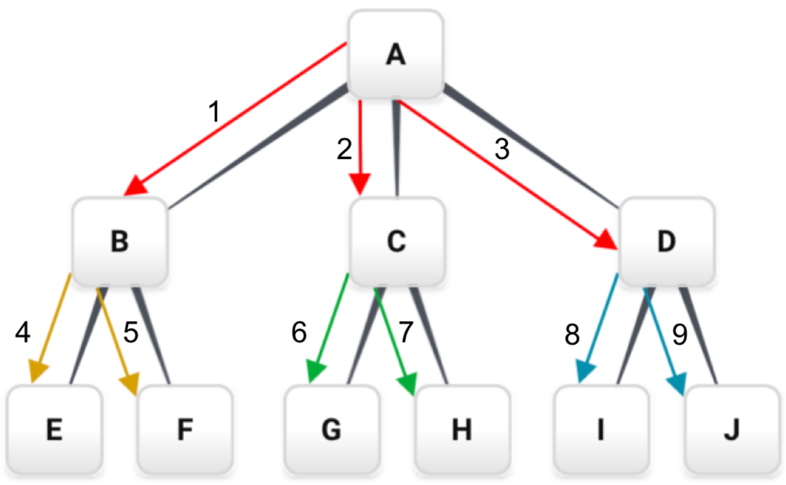


Figure 4.5: Breadth First Search

which are marked with the red arrows. The first node that will be visited, will be B, with a distance d_{AB} from A of 7 units, as this path is from the origin, it is not required to add any previous distances, then C with $d_{AC} = 5$ and then D with $d_{AD} = 2$. The first element in and out of the queue will be B, making it the next active node. The next step will be to visit the children of B, identified with the yellow arrows. As the goal is to identify traveling to which leaf corresponds to the shortest path from A, the distance to the children of B will be the distance from A to B in addition to the distance from B to each node. For these nodes, the distances will be $d_{ABE} = 11$ and $d_{ABF} = 13$). These two nodes are also added to the queue, which now has nodes $\{C, D, E, F\}$. The next node to explore will be C and after that D. At this point, the queue contains nodes $\{E, F, G, H, I, J\}$, which will continue to get popped out from the list, but as all of them are leaves, without children, no new nodes will be explored and once the queue is empty, the calculation will be completed. The order in which the tree was explored is $\{A, B, C, D, E, F, G, H, I, J\}$.

In the end, all distances will be calculated, which were reported in Table 4.1 and the shortest path will be found, but by having to visit every node, this approach is not attractive for more complex scenarios with a larger number of possibilities, as it will have to traverse the complete tree to be able to select the best option.

Depth First Search

The next way of traversing a tree structure to find the shortest path is given by the **depth first search** approach. In this case, one of the branches originated in node A is selected and the whole depth of this path will be traversed, this means visiting all of its descendants. Once the bottom of that branch is reached, the algorithm backtracks and explores other branches that it had found on the way. When all elements from that section are visited, it backtracks all the way back into the root, repeating the same steps on another branch, until all possible paths and nodes are explored. Similarly to **breadth first search**, to keep track of the visited nodes and decide which element to evaluate next a data structure is necessary, in this case, a stack is used. Every node that gets popped out of the list is gets verified to see if it has untraveled branches and if it has, one of those candidates will be explored next.

The current approach to the example is depicted in Figure 4.6. As in the previous case, the numbers next to the arrows tell the order on which the paths were explored. The colored arrows group through similar colors the order followed, having the same exact color for paths that were visited one after the other and on a darker tone the path visited after backtracking.

In this example, from A it would add B, C and D to the stack. From there, it would pop the element at the top of the stack, in this case B and explore its branches and add the nodes reached through them to the stack. This means, updating the stacking to $\{E, F, C, D\}$. While doing this operations, the distances are being updated on each respective node, so for the A case, $d_{AB} = 7$, $d_{AC} = 5$ and $d_{AD} = 2$, then following the order, $d_{ABE} = 11$. When E is popped, due to it being a leaf, there are no more descendants or nodes to explore, so the next step is to backtrack to B, or in the current stack $\{F, C, D\}$, pop the next term F and repeat. After the whole group of descendants of B are visited, thus removed from the stack, it is necessary to continue with the

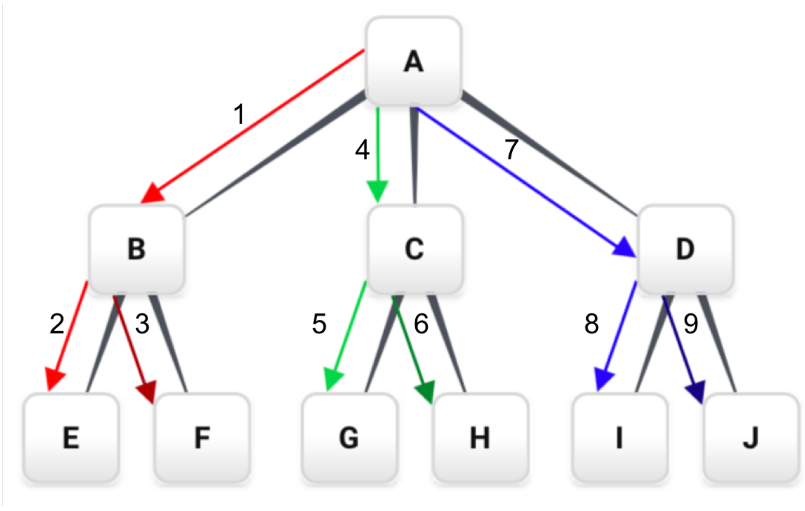


Figure 4.6: Depth First Search

next element at the top, which is *C*. The same steps are repeated until there are no more nodes in the stack, at which point the complete tree has been explored. The order for choosing the next element between the group of neighbors could have been different and instead of going first to *B* in an alphabetical order, *C* or *D* could have been chosen, but in the end this does not reduce the number of steps required to find the shortest path. As in **breadth first search**, to find the solution it is necessary to visit the complete tree, obtain all the distances and compare them to find the shortest path.

Best First Search

Also called Informed Search, the final group of algorithms is **best first search**. The main characteristic of this group is that the algorithms guide the exploration based on an evaluation of the possible nodes and the most favorable one is selected. The determination of how favorable a node is, is determined by a rule that governs the execution of the algorithm. Different rules give rise to different algorithms, which could for the same problem arrive at the solution through different approaches; so it is important to match the problem with an algorithm which exhibits an adequate performance in its field.

From the definition itself it is possible to notice that these algorithms should reduce the number of nodes that need to be explored to find the shortest path. This reason makes them more attractive for the Test Flow Optimization problem, so the two algorithms that adapt better to the problem's data will be discussed.

Dijkstra's algorithm Initially proposed to find the shortest path between two nodes, it has been extended to be capable of finding the shortest path from the source to all other nodes.

The main characteristic of this algorithm is that it guides the search by following the path with the lowest value to travel from the current to the next element. The process of solving a problem through this algorithm utilizes two separate data structures, one is the visited list and the other is the unvisited list, where all nodes start in and their distances are set to infinity or unknown. The following steps are followed:

1. Select the source node as the current node.
2. For all unvisited neighbors of current node, calculate the distance from source through current node and update their distance value only if it is smaller than a preexisting value.
3. Move current node to visited list. Once in the visited list, a node will not be checked again.
4. From the unvisited list, select as current node, the one with the lowest value.
5. Repeat steps 2 and 3 until destination node has been marked as visited.

For the example case, the destination node is any node from the third level of the tree, so if one of these is selected as current node, the algorithm has found the shortest path.

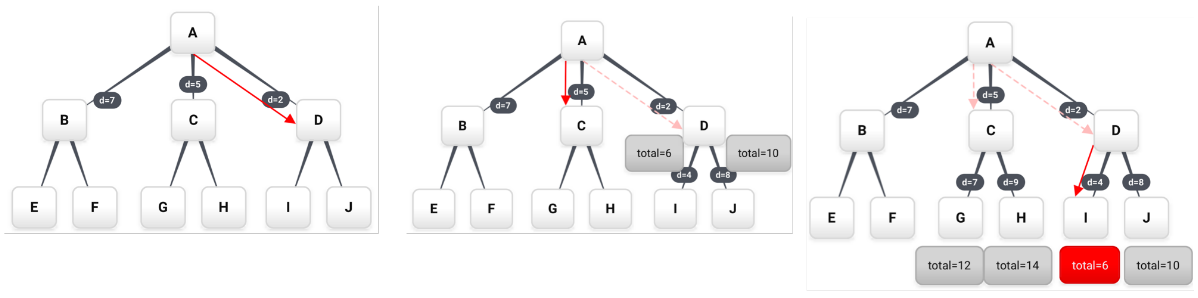


Figure 4.7: Dijkstra's Algorithm

Figure 4.7 shows a condensed visual representation of the steps required to find the shortest path for the example case. In (a), the root of the tree is the current node, so all three of its neighbors are explored and the one with the shortest distance selected (Node D with $d = 2$). After setting D as current node and exploring its neighbors I with $d = 6$ and J with $d = 10$ and comparing against all other unvisited nodes, the lowest distance does not belong to one of the most recently calculated elements, but to node C with $d = 5$, which is shown in (b), so the algorithm backtracks to A and visits C. When calculating the distances for C and comparing them with the other unvisited nodes, the lowest value is found, Figure 4.7 (c) at I, which is even lower than node B, so it is selected as the next node and because it is one of the possible destination nodes, this means that the shortest path is found.

The number of nodes that were explored including the root is 8, node A, its 3 children and the 2 children of C and 2 of D. Comparing to **breadth first search** and **depth first search**, it explored 2 fewer cases. In this scenario the improvement is not that significant, only a 20% but in a larger search space with more levels on the tree and nodes, this difference can be increased greatly, finding the solution faster by not having to explore all the nodes and paths.

A* algorithm The final option based on the classification of relevant algorithms. It is based on **Dijkstra's** algorithm, with the addition of a heuristic to help guide more accurately the search. It's approach is very similar to **Dijkstra's**, following the same steps to determine which node to explore next. From a list of unvisited nodes it selects the most promising one based on the lowest cost, with the only difference that in this case, the cost is made up of two parts, instead of only one.

Equation 4.1 refers to the cost at a node n, with $f(n)$ corresponding to the cost, $g(n)$ to the path from the initial node to node n and $h(n)$ the heuristic, which is the differentiating characteristic of this algorithm. The heuristic is based on a prediction of the remaining effort to travel from n to the destination node. With the use of the heuristic, an additional weight is given to the nodes, aiding in the search of the shortest path, by considering not only the previous steps taken, but also the proximity to the destination. To guarantee that the found path is the shortest, the heuristic must be admissible, which means that it must not overestimate the cost of reaching the destination.

$$f(n) = g(n) + h(n) \tag{4.1}$$

Based on the heuristic's definition it can be concluded that this method is destined for shortest path between two points problems. The reason for this is that to formulate a general admissible heuristic, it is necessary to have a clear end point, or else it will not be possible to guarantee that a found path is the shortest one.

When comparing this algorithm with **Dijkstra's**, the relation between the two of them becomes evident. **Dijkstra's** algorithm can be seen as an implementation of **A*** where the h values for all nodes are set to 0, this guiding the search based only on the traveled path.

Due to the heuristic formulation, the current example and the test flow optimization do not strictly adhere to **A*** requirements by cause of not ending at a single point. The effect of this is that it is not possible to determine an admissible $h(A)$, as each branch has different end points, thus different distances to them.

4.1.2. Modified Dijkstra

After exploring the different algorithms and seeing how they conform to the **test flow optimization** scenario, the algorithm that will be implemented is based on Dijkstra's, borrowing some elements from A*.

Table 4.2: Comparison of analyzed algorithms

Algorithm	Advantages	Disadvantages
Breadth First Search	-Straightforward formulation -Finds shortest path -Single source/multiple destinations	-Needs to explore the complete space to find the solution
Depth First Search	-Straightforward formulation -Finds shortest path -Single source/multiple destinations	-Needs to explore the complete space to find the solution
Dijkstra's	-Single source/multiple destinations -Typically converges to solution faster	-More complex formulation than previous two
A*	-If heuristic is admissible and good, finds shortest path the fastest	-Single source/single destination -Most complex, based on additional difficulty of finding admissible heuristic

The **modified Dijkstra** implementation consists as mentioned in Section 4.1, of creating a tree representation that includes the original case as the root node and on the nodes connected to the root through the branches, create all possible combinations of test moments that the user is willing to try and compare to find the test flow with the lowest cost. This tree will be traversed following **Dijkstra's** approach, which is to select the next node from all available nodes, based on which node entails the lowest cost to reach it. The modification to the algorithm can be seen more as an expansion into the **A*** model, where a second parameter is involved in the cost determination. Due to the nature of the data being represented in the nodes, it is not possible to represent the second parameter independently. This element exists in all non end node elements and it corresponds to the inclusion of all yet unexplored test moments.

Another important characteristic of the tool's implementation is that it does not only consider the cost, it is also capable of exploring different cases based on the number of escapes, thus two sets of results are possible. The first one is finding the test flow with the lowest cost, for a given number of escapes. The second one being finding the test flow with the lowest number of escapes, for a given cost.

Based on the the possibility of guiding the search through lowest cost or lowest escapes, the previously mentioned unexplored test moments must be taken into consideration. These unexplored test moments can have both a fault coverage and a test cost, so a rule must exist to handle them. During optimization based on cost, the option with the lowest coverage and lowest cost will be used, producing cheap results due to the low cost of the testing and also to not detecting as many defective elements, thus not needing to compensate for the bad dies by manufacturing more units. In the escapes optimization case, the highest fault coverage option will be used with the lowest cost, thus ensuring the lowest number of escapes at the incomplete test moments, but without over estimating the cost. By doing this, it is possible to ensure that the parent node will always have the lowest value between him and all of its descendants. This property allows us to conclude that if an end node γ is discovered and all other unvisited nodes have a higher cost than γ, there will not be a test flow with a lower cost than the one corresponding to γ, thus γ is the test flow with the lowest cost.

A different example from the previous one will be used to illustrate the process by which the algorithm finds a solution. The new example consists of a stack of 4 dies achieved through 3 linear stacking operations. In the first one die 2 is stacked on top of die 1, in the second one die 3 is stacked on top of the stack from operation one and in the third stacking operation die 4 is stacked on top of the partial stack. The other details from this case are not necessary to follow the example.

The user wants to find the test flow with the lowest cost obtained from modifying 3 test moments, while only accepting solutions with fewer test escapes than 9000 DPPM. The 3 test moments can be seen in Figure 4.8, the first test moment is specified by the first two lines and it consists on exploring three cases for the pre-bond test of die 2. As every test moment has both a cost and a fault coverage related to it, both parameters are defined, the first line mentions the costs and the second one the fault coverages. The second modification to the test flow to is change the test performed to find defects in die 2 after the first stacking operation, seen in the third and fourth lines. For this case, only two cases will be explored. The last scenario is applied to die 3 after the second stacking operation, where also two cases will be explored.

The tree created based on the previously mentioned parameters is shown in Figure 4.9. Node 0 represents the original case and all other nodes correspond to a combination of the test moments being explored. The tree has four levels, with the first one only containing node 0, the additional three levels correspond to the

```
TF_T_cost_die_prebond[2]                      1.25 1.75 2.00
TF_T_fc_die_prebond[2]                        0.90 0.96 0.98

TF_T_cost_die_stackOp[1][2]                   0.75 1.25
TF_T_fc_die_waferDefects_stackOp[1][2]        0.94 0.98

TF_T_cost_die_stackOp[2][3]                   1.25 1.75
TF_T_fc_die_waferDefects_stackOp[2][3]        0.95 0.99
```

Figure 4.8: Test moments explored in example

three test moments mentioned before. The three nodes in the second level are the three different values explored for the first test moment.

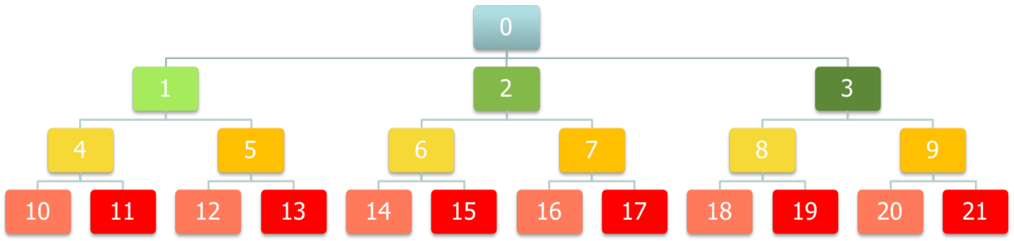
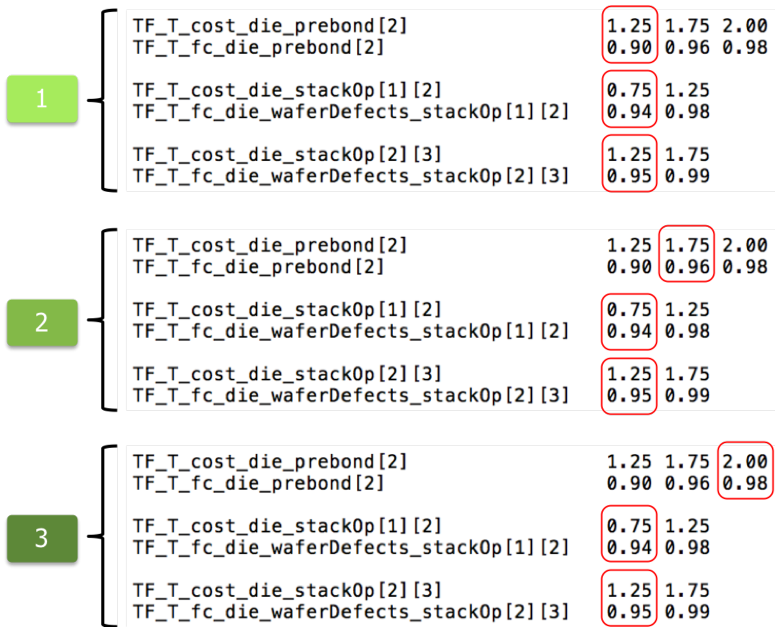


Figure 4.9: Test moments explored in example

Node 1 and its descendants will have in all of their cases, the first option for the first test moment , while modifying the other options based on the position of the node in the tree. Furthermore, as only the bottom level represents the complete test flows, in the previous levels the optimal cases are used for the test moments which have not been set yet. Figure 4.10 illustrates this. The three nodes shown are the ones from the second level, for the first test moment, each node selects a different cost and fault coverage pair, with node 1 selecting the first pair, node 2 the second and node 3 the third. The values for the other two test moments are chosen based on the rule defined for the optimization targeting cost, thus lowest cost and lowest fault coverage.



```
TF_T_cost_die_prebond[2]                   [✓ 1.25] 1.75 2.00
TF_T_fc_die_prebond[2]                     [✓ 0.90] 0.96 0.98

TF_T_cost_die_stackOp[1][2]                [✓ 0.75] 1.25
TF_T_fc_die_waferDefects_stackOp[1][2]     [✓ 0.94] 0.98

TF_T_cost_die_stackOp[2][3]                [✓ 1.25] 1.75
TF_T_fc_die_waferDefects_stackOp[2][3]     [✓ 0.95] 0.99
```

```
TF_T_cost_die_prebond[2]                   1.25 [✓ 1.75] 2.00
TF_T_fc_die_prebond[2]                     0.90 [✓ 0.96] 0.98

TF_T_cost_die_stackOp[1][2]                [✓ 0.75] 1.25
TF_T_fc_die_waferDefects_stackOp[1][2]     [✓ 0.94] 0.98

TF_T_cost_die_stackOp[2][3]                [✓ 1.25] 1.75
TF_T_fc_die_waferDefects_stackOp[2][3]     [✓ 0.95] 0.99
```

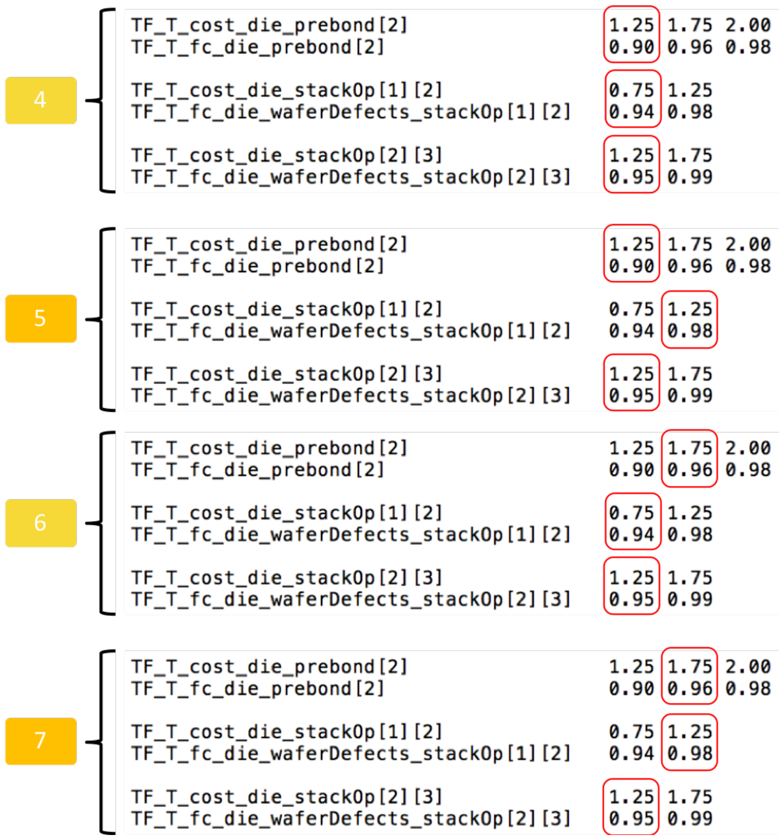
```
TF_T_cost_die_prebond[2]                   1.25 1.75 [✓ 2.00]
TF_T_fc_die_prebond[2]                     0.90 0.96 [✓ 0.98]

TF_T_cost_die_stackOp[1][2]                [✓ 0.75] 1.25
TF_T_fc_die_waferDefects_stackOp[1][2]     [✓ 0.94] 0.98

TF_T_cost_die_stackOp[2][3]                [✓ 1.25] 1.75
TF_T_fc_die_waferDefects_stackOp[2][3]     [✓ 0.95] 0.99
```

Figure 4.10: Values used for the nodes of the second level of the tree

Figure 4.11 illustrates the test moment conformation for four of the cases in the third level. In this level, there are only two options, so the respective test moment will only change between those two. The tone of the colored boxes is used to highlight this, where nodes 4 and 6 (same tone) have the same values for the second test moment, but different ones for the first test moment based on the node's parent. This also occurs with nodes 5 and 6 and can be extended to every level from the tree. The tone denotes the values used for the test moment in question, matching the values in similarly matched toned nodes.



```
4
TF_T_cost_die_prebond[2]                        1.25 1.75 2.00
TF_T_fc_die_prebond[2]                          0.90 0.96 0.98

TF_T_cost_die_stackOp[1][2]                     0.75 1.25
TF_T_fc_die_waferDefects_stackOp[1][2]          0.94 0.98

TF_T_cost_die_stackOp[2][3]                     1.25 1.75
TF_T_fc_die_waferDefects_stackOp[2][3]          0.95 0.99

5
TF_T_cost_die_prebond[2]                        1.25 1.75 2.00
TF_T_fc_die_prebond[2]                          0.90 0.96 0.98

TF_T_cost_die_stackOp[1][2]                     0.75 1.25
TF_T_fc_die_waferDefects_stackOp[1][2]          0.94 0.98

TF_T_cost_die_stackOp[2][3]                     1.25 1.75
TF_T_fc_die_waferDefects_stackOp[2][3]          0.95 0.99

6
TF_T_cost_die_prebond[2]                        1.25 1.75 2.00
TF_T_fc_die_prebond[2]                          0.90 0.96 0.98

TF_T_cost_die_stackOp[1][2]                     0.75 1.25
TF_T_fc_die_waferDefects_stackOp[1][2]          0.94 0.98

TF_T_cost_die_stackOp[2][3]                     1.25 1.75
TF_T_fc_die_waferDefects_stackOp[2][3]          0.95 0.99

7
TF_T_cost_die_prebond[2]                        1.25 1.75 2.00
TF_T_fc_die_prebond[2]                          0.90 0.96 0.98

TF_T_cost_die_stackOp[1][2]                     0.75 1.25
TF_T_fc_die_waferDefects_stackOp[1][2]          0.94 0.98

TF_T_cost_die_stackOp[2][3]                     1.25 1.75
TF_T_fc_die_waferDefects_stackOp[2][3]          0.95 0.99
```

Figure 4.11: Values used for the nodes of the third level of the tree

Moving on to the execution of the algorithm, it follows the same rules as in **Dijkstra's** case, where the next node is the cheapest one, but due to the acceptable escapes figure, at every node it also verifies if it complies with this restriction, if it does not, then that node is invalidated and will not be explored further, as it is not possible to get a better result further down in the tree.

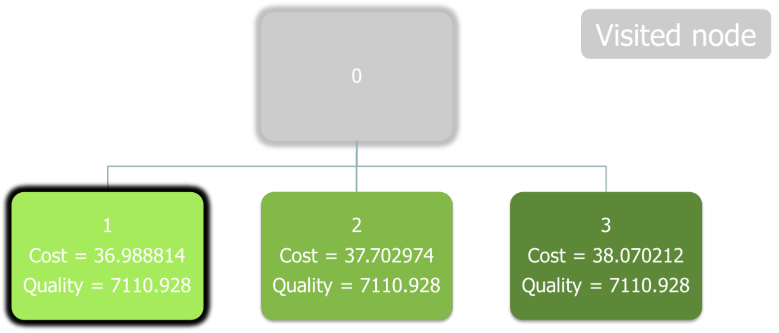


Figure 4.12: First step: exploring the children of the original node

The first step is to explore the children of the original case, this is shown in Figure 4.12, where the cost and quality(escapes in DPPM) for each node is written. Here, all three cases conform with the escapes requirement and the cheapest option is given by node 1 so it selected as the next node to be explored.

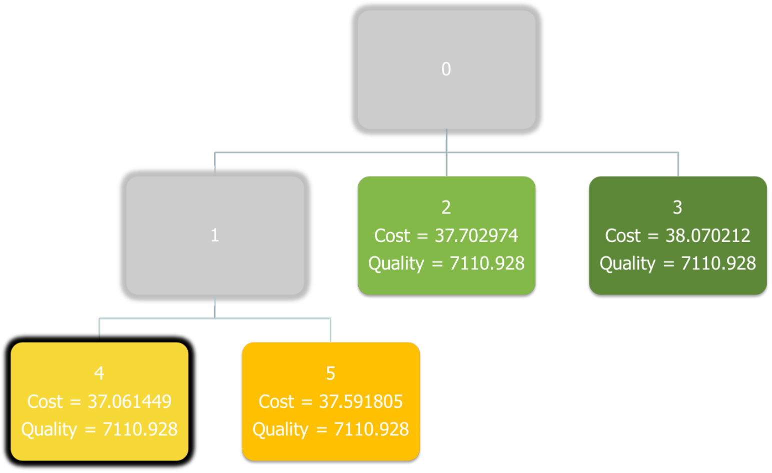


Figure 4.13: Second step: exploring node 1's children

In Figure 4.13, the children of node 1 are discovered and node 1 is grayed out and moved into the visited list. At this point, all nodes but the gray ones are part of the unvisited list, so they are candidates to be selected. The node with the lowest cost is 4, so it is selected and will be expanded next.

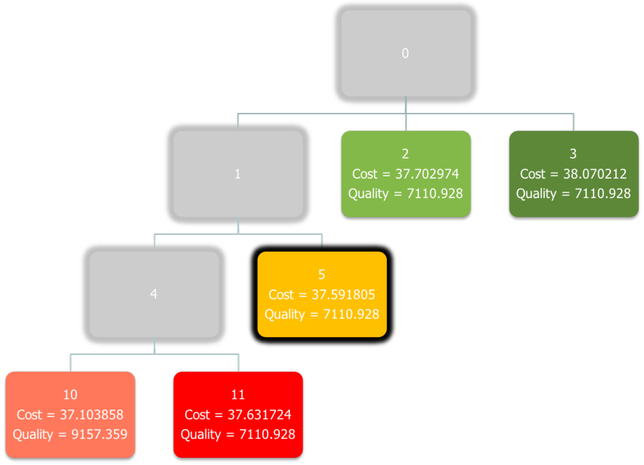


Figure 4.14: Third step: exploring node 4's children

When node 4's children are explored, a new situation arises, the cheapest node from all of the unvisited list violates the maximum acceptable escapes number, thus it can not be chosen and it gets invalidated. The next cheapest option is then selected, with it being node 5. This situation is shown in Figure 4.14

The final step seen in Figure 4.15, occurs after expanding node 5 and realizing that the cheapest node in the unvisited list corresponds to node 11, which is an end node (located in the lowest level of the tree) and it adheres to the escapes rule, thus it is selected as the cheapest solution for this case.

In the explored example, it was necessary to explore 9 nodes (the original case is not necessary as it is automatically calculated by 3D COSTAR), if this case would have been solved via the other traversal methods, the complete tree (21 nodes) would have been necessary to find the cheapest option, so in this case only 43% of the cases were needed to find the best case. If comparing against a pure exhaustive search case, there is still an advantage. In a pure exhaustive search, only the nodes of the bottom level with the complete test flow will be explored, so the results for those 12 nodes would be necessary. So comparing **modified Dijkstra** against exhaustive search, the answer was found while only visiting 25% fewer elements.

To expand on the previous point, the following equations describe the best case and the worst case scenarios for the **modified Dijkstra** algorithm.

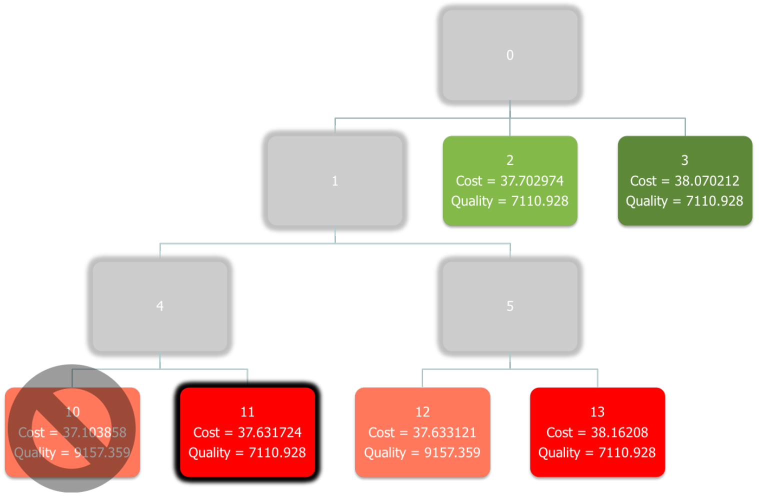


Figure 4.15: Final step: Finding the cheapest solution

For a tree t with L elements per level and n levels

$$\text{Best case} = \sum_{i=1}^{n} L_i \tag{4.2}$$

Equation 4.2 refers to the best case scenario, which occurs when the path of explored nodes does not only goes through one node per level, thus the solution is found by having to explore only one branch from the root. In the example, this would have been the case if the escapes restriction had a looser figure, thus allowing node 10 to be selected as the solution. This scenario would have been solved by visiting 7 nodes in contrast to the 9 in the actual solution.

$$\text{Worst case} = (\sum_{i=1}^{n} \prod_{j=1}^{i} L_j) - 1 \tag{4.3}$$

The worst case scenario is given in Equation 4.3. The worst case occurs when all of the tree nodes need to be explored, thus finding a solution in the same time as **breadth first search** or **depth first search**. Compared to exhaustive search, this scenario is worse because in exhaustive search only the bottom nodes need to be explored, but cases that match worst case behavior are not typical and as trees get bigger, a faster path tends to be found faster.

4.2. Stacking Order Analysis

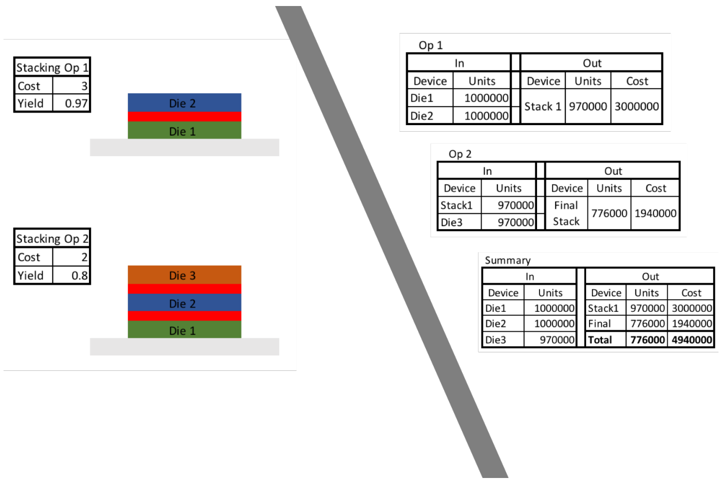
The next optimization concerns the stacking order. This optimization examines the possibility of finding an alternative order for the stacking operations that can lower the cost producing a certain SIC.

3D COSTAR creates a representation of the stack based on the specified stacking operations. It is possible that while keeping the same stack structure, altering the order of the stacking operations will reduce the cost. Although in practice the cost is dependent upon many factors and it is not advisable to use just one element in isolation to predict the behavior of the model, the following basic example will be used to illustrate how altering the order can lower the cost.

The example is comprised of two similar cases, where the only difference will be the order of the stacking operations. All cost and yield parameters will remain equal. The example consists of a 3D stack, of three dies and two stacking operations. Die manufacturing cost and yield are assumed as equal for all three devices, eliminating the effects caused by manufacturing more of one die than the others. Testing is considered at 100% without having any additional cost.

Figure 4.16 shows the first case. The first stacking operation consists of die 2 on top of die 1. This operation has a higher cost and a higher yield than the other operation. The second operation will produce the final stack, it consists of stacking die 3 on top of the partial stack. This operation has a lower cost and a higher propensity to introduce defects, thus a lower yield. Due to the order, the expensive operation is performed first, 1,000,000 times, incurring in a cost of 3,000,000 and obtaining 970,000 good elements of stack 1. Those

970,000 stacks are used as the base die for stacking operation 2, so the same number of units of die 3 will be required. After the second stacking operation, 776,000 good final stacks remain, this step costs 1,940,000. The total for this case will be 776,000 final stacks with a cost of 4,940,000.



Stacking Op 1	
Cost	3
Yield	0.97

Stacking Op 2	
Cost	2
Yield	0.8

Op 1

In		Out		
Device	Units	Device	Units	Cost
Die1	1000000	Stack 1	970000	3000000
Die2	1000000			

Op 2

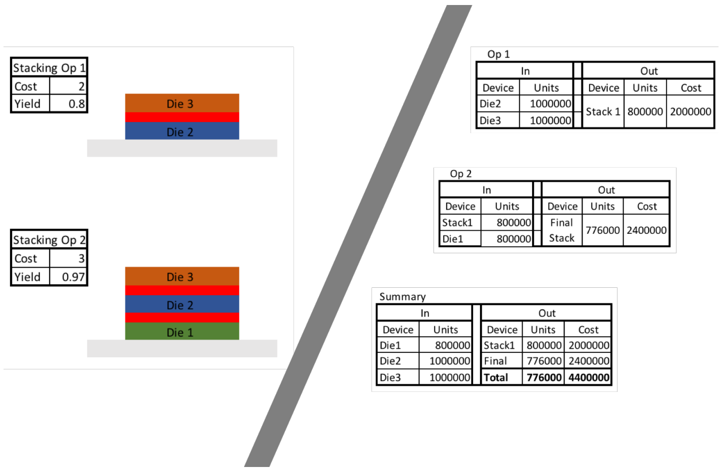
In		Out		
Device	Units	Device	Units	Cost
Stack1	970000	Final Stack	776000	1940000
Die3	970000			

Summary

In		Out		
Device	Units	Device	Units	Cost
Die1	1000000	Stack1	970000	3000000
Die2	1000000	Final	776000	1940000
Die3	970000	**Total**	**776000**	**4940000**

Figure 4.16: Stacking order, first case

The second case seen in Figure 4.17, presents the scenario where the order of the stacking operations is swapped. In this case, the cheap operation with its low yield is performed first and the expensive one performed last. Following this order, after the first stacking operation 800,000 good partial stacks exist, with a cost of manufacturing them of 2,000,000. Those 800,000 are used as input for the second operation with the same amount of die 1, producing 776,000 final stacks with a cost of 2,400,000 for the second operation and a total cost of 4,400,000.



Stacking Op 1	
Cost	2
Yield	0.8

Stacking Op 2	
Cost	3
Yield	0.97

Op 1

In		Out		
Device	Units	Device	Units	Cost
Die2	1000000	Stack 1	800000	2000000
Die3	1000000			

Op 2

In		Out		
Device	Units	Device	Units	Cost
Stack1	800000	Final Stack	776000	2400000
Die1	800000			

Summary

In		Out		
Device	Units	Device	Units	Cost
Die1	800000	Stack1	800000	2000000
Die2	1000000	Final	776000	2400000
Die3	1000000	**Total**	**776000**	**4400000**

Figure 4.17: Stacking order, reordered

As can be seen, both cases produce the same number of good stacks, but the total cost differs by 540,000, with the second case cheaper than the first. This is due to having to perform fewer of the expensive operations. In the first case, the expensive operation occurs 1,000,000 times and then the cheap operation is performed only to the output of the expensive one. In the second case, by having first the low yield operation, the largest waste in terms of defective dies occurs initially, reducing the number of times the expensive operation will be performed, thus contributing less to the cost. It should be noted that this is an artificial case used to illustrate the situation. Although in this case, having the expensive operation in the end resulted in a lower cost, this can not be generalized and assumed that this will always occur, therefore implementing this methodology in 3D COSTAR is very valuable, as it will create the scenarios and calculate their costs, allowing the user to find an optimal stacking order based on lower cost.

4.3. TSV Manufacturing Cost

As mentioned in the **manufacturing** section in chapter 3, the wafer processing is comprised of two elements, the yield for manufacturing the dies and the yield for the interconnect. Nevertheless, the cost is only considered in one general parameter, the wafer manufacturing cost.

Dies and interconnect, differ in terms of the manufacturing process, even to the point where it is possible that a fabrication house in charge of producing the dies on the wafers, does not posses the technical capabilities to create the TSVs, so a separate specialized fabrication company needs to be involved in that part of the process. For the previous reason, it is valuable to separate from the wafer manufacturing cost, the interconnect or TSV manufacturing cost, providing a more detailed breakdown of the costs involved in the first stages of the manufacturing process.

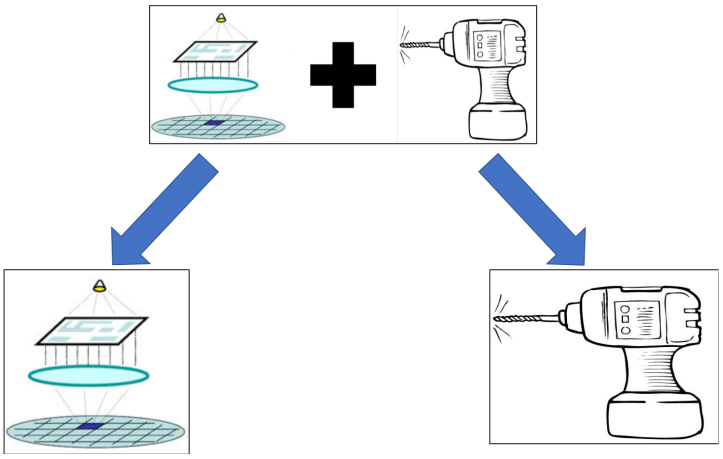


Figure 4.18: Separation of wafer processing costs

Figure 4.18, illustrates this scenario, originally, both parts of the process where considered as one, but after applying the proposed change, they will be separate elements, creating a one to one relation between yield parameters and cost parameters.

5 Simulation Results

This chapter discusses the implementation details of the extensions to the 3D COSTAR tool, which were explained in the previous chapter. It concludes by proposing three case studies each applying to a different stacked IC scenario, which are analyzed by the tool and their results presented. Section 5.1 covers the implementation details for 3D COSTAR and the extensions. Section 5.2 presents the three case studies as well as their results.

5.1. Implementation

Since the objective was to extend the 3D COSTAR tool with the extensions discussed in chapter 4, the first step consisted of reviewing the code of the original version. With this, it was possible to understand how the complex cost modeling process was being performed.

The language used to write the tool is C. As can be seen in Figure 5.1, twenty files make up the internal structure of 3D COSTAR. From those twenty, nine are header files tasked with specifying the declaration of functions utilized in their corresponding source files, as related by their similar name. From the remaining eleven files, one is the main file, five correspond to the cost classes (Design, Manufacturing, Test, Packaging and Logistics). Next is sensitivity.c, which implements the sensitivity analysis, lib.c has auxiliary functions for array operations as sum and dot product, read_io.c handles the data scanning from the input file, along with the printing of arrays to the results files, top.c contains the functions which execute the cost model's equations and excel.cpp is used to export the data to excel to make visual representations of the result.

To organize the data, structs are utilized. Each cost class will have its own struct, where the costs, yields and all other information is stored. Due to the difference between classes, the struct's composition will vary greatly, but in essence, all of them serve the same basic function, to store in an organized way the data that characterizes their particular class, making it available to be later used during the cost calculation.

The main function guides the execution of the program. It begins by declaring pointers for each class' struct, an additional struct in charge of handling files and another one to collects results from the calculations. After that, initialization functions from each class will be called as well as for sensitivity, where it is checked if the analysis will be performed. In these functions, the struct's variables will be initialized with the correct dimensions for the particular case. The variable contents will be assigned by reading the input file, through the use of the read_io functions. Once all classes have prepared their data, the cost will be calculated by the functions from the top file. The output file is then printed with the results.

After calculating the cost, the next task in the main file is to verify if the user requested for the creation of a debug file. The debug file contains the utilized data, intermediate calculations and detailed outputs. To create the debug file, two functions are utilized. The first one reproduces the values used to set the experiment and the second one accesses the records stored in the results collector to print the detailed version of the output. At this point, the default experiment is finished and all that remains is to perform the sensitivity analysis and free the used memory.

The sensitivity analysis works by reading specific keywords related to this process. Those keywords specify the different scenarios that the user would like to explore. For those keywords, a range of values is specified as well as a step size, which is used to calculate the values to be input between inside the range. To run these modified cases, the keywords related to sensitivity are read and then matched with their respective memory location in the original cost classes. Once matched, the new values are plugged into the memory location

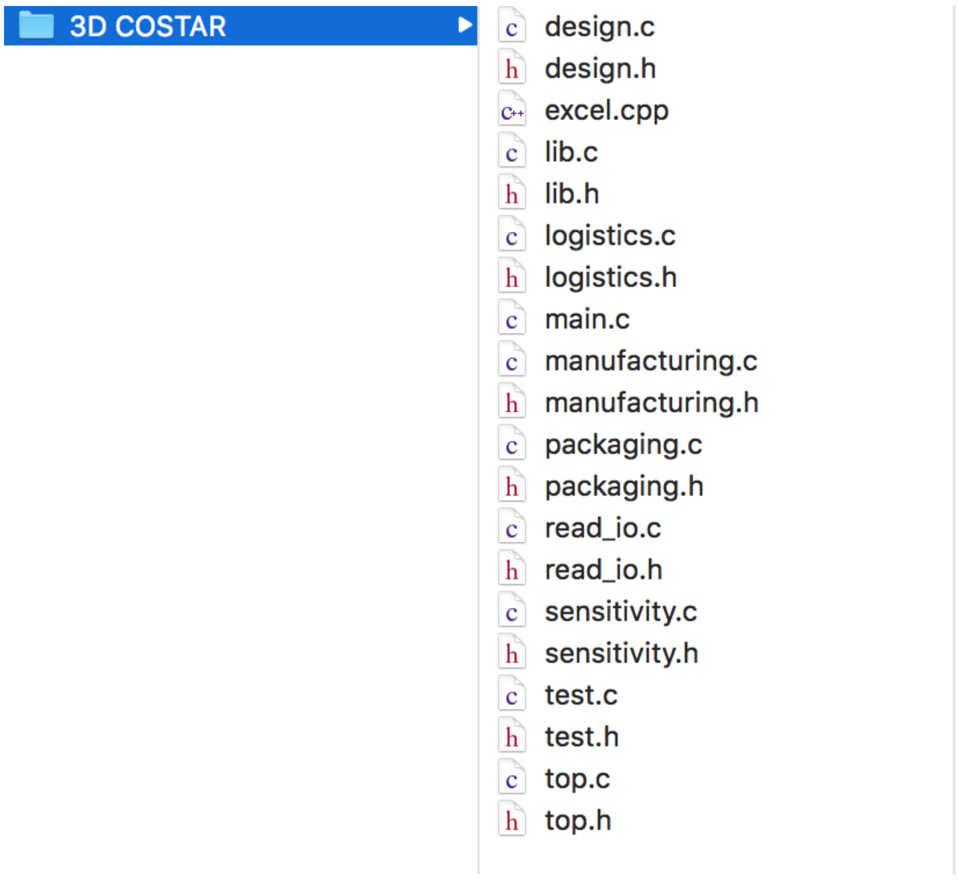


Figure 5.1: Files conforming 3D COSTAR

and the cost is recalculated. The result is stored on a standard output file generated for every sensitivity experiment, as well as a debug file and a summary file, which presents the total cost, escapes and cost of each of the five classes for all the sensitivity experiments in one place.

Once the code was reviewed certain modifications were carried out. These initial modifications were conducted to improve the general operation of the tool, but were not related to the extensions. For the extensions, a new set of modifications were later performed, which will be explained during each extension's implementation details. The following list highlights a select number of modifications performed to the original 3D COSTAR tool.

- Reimplemented functions used to read the data from the input file, to enable the handling of cases were a parameter is not specified. In those cases, a default value is used instead.
- Added functions to handle array printing to the output files.
- Cleaned the output files, as information was being repeated and erroneous text was being printed.
- Separated the LINUX version of the tool to handle output file creation differently for when running on a server or on a client.
- As the sensitivity analysis modifies the original case to recalculate cost based on different parameters, created functions to copy the original case and restore it once the sensitivity analysis is done. This prevents data contamination for further cases, as the parameters from the original case are returned to their memory locations.
- Created checkpoints to prevent unnecessary allocation of memory and freeing of unallocated elements. For example, the sensitivity memory space was being allocated before confirming the existence of the sensitivity analysis. Now it only allocates and frees it if the analysis is performed.

5.1.1. Test Flow Optimization

To implement the **test flow optimization** a source file and its corresponding header file were added to 3D COSTAR. Figure 5.2 presents a flowchart which describes the implementation. To achieve this, a total of 25 functions are utilized in the modified Dijkstra's implementation and a struct with 36 elements which organizes and contains all the variables needed along the optimization code. A list of 17 valid keywords that covers the complete testing options is used to compare the input parameters and use those values during the cost calculation.

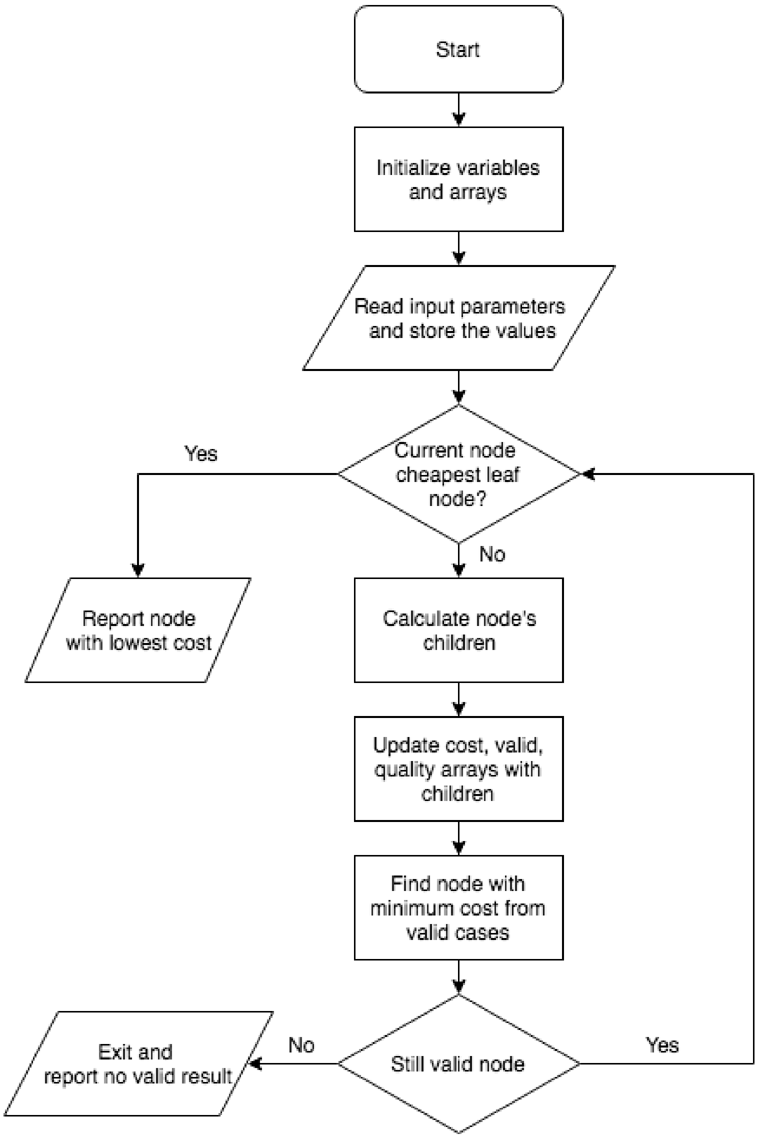

Start
Initialize variables and arrays
Read input parameters and store the values
Current node cheapest leaf node?
Yes
No
Report node with lowest cost
Calculate node's children
Update cost, valid, quality arrays with children
Find node with minimum cost from valid cases
Still valid node
No
Yes
Exit and report no valid result

Figure 5.2: Test Flow Optimization Flowchart

The code can be divided into two large sections based on the functionality that they provide,. The first one responsible for the initialization of the environment and the second one in charge of the execution of the algorithm.

Initialization of the Test Flow Optimization environment

The following functions are tasked with memory allocation, data reading and preparation to enable the correct execution of the algorithm.

Function initialize_tf_optimization

This function is called by the *main* routine. Its first task is to verify if the implementation will be performed or not. When indeed it is performed, it continues by reading the parameter which specifies how many keywords or input lines will be read and into how many groups they are separated. Next, it checks if the optimization is based on cost or escapes and also the threshold specified. Following that, it proceeds to call the *allocate_tf_opt_parameters*, followed by *assign_tf_opt_keyword_init_ptrs*, and *decode_tf_opt_parameters*. At this point, there is enough accessible data to calculate the number of leaf nodes and the total size of the tree, which will become useful later as the experiment space will be allocated. To find the number of leaf nodes, Equation 5.1 is used and for the total size, Equation 5.2.

For a tree t with L elements per level and n levels

$$\text{Leaf nodes} = \prod_{i=1}^{n} L_i \tag{5.1}$$

$$\text{Total size} = (\sum_{i=1}^{n} \prod_{j=1}^{i} L_j) \tag{5.2}$$

After calculating the sizes, functions *check_tf_opt_indices*, *allocate_tf_opt_experiments_parameters* and *copy_original_value_tf_opt*, which backs up the original case data, are called and at that point, the current function, *initialize_tf_optimization* finishes.

Function allocate_tf_opt_parameters

The majority of the allocations is performed in this function. It sets up all the variables needed to read the specified keywords as well as the values they will acquire during the different experiments. At this stage, there is a memory location assigned to each of the values read from the input file. Other variables that help identify the data type and memory location of the actual cost class variable to which it corresponds, are also assigned.

Function assign_tf_opt_keyword_init_ptrs

In this function, a list with all 17 possible keywords is created and for each of them, the data type and how many dimensions it has, is specified as well as the pointer location of the original case. By dimensions it is understood if the keyword arguments consist of a one dimension array or a matrix. For example, the fault coverages of the pre-bond phase are a one dimension array, as each element corresponds to a die and for each die there is only one fault coverage being used. However, during the mid-bond phase there might be more than one possible test moment, so one dimension specifies to which stacking operation the argument belongs to and the other dimension to which specific die.

Function decode_tf_opt_parameters

After setting the memory, it is time to read the keywords and their values. Due to the nature of the input, it is easier to break down the parsing into four functions including this one, each one in charge of performing a different task. *decode_tf_opt_parameters* begins the process by identifying the lines in the input file that contain the **test flow optimization** parameters with the help of an identification keyword. These lines are read entirely and passed on to *decode_tf_opt_single_entry*, which then separates each word and values from the read line to their respective variables. After that, *match_tf_opt_token* with the help of *match_tf_opt_single_kw* will match the read keyword with one of the 17 options and continue by taking the pointer location from the list created in *assign_tf_opt_keyword_init_ptrs* and assigning it to the read keyword, as well as specifying if the type is cost or fault coverage. Once the lines are processed in their totality, the default case for each keyword is identified, based on the type of keyword (FC or cost), as the rules differ between the two of them and also depending on if the optimization is performed for lower cost or for lower escapes.

At the current stage, all lines from the input file related to the optimization have been processed, the keywords identified and the values stored, making them available to later be replaced in the memory locations of the test cost class to find the cost for the new cases.

Function check_tf_opt_indices

Back in *initialize_tf_optimization*, the next step is to verify the previously read data. This task is performed by *check_tf_opt_indices*, which checks that the dimensions of the read keywords correspond to existing elements from the manufacturing class. In other words, this function makes sure that the optimization keywords are applied to existing stacking operations, dies and interconnects, thus preventing for example a case where on a three die, two stacking operation stack, the user tries values for a third stacking operation or for a die which was not specified in the manufacturing parameters.

Function allocate_tf_opt_experiments_parameters

After the verification and with the confirmation that the data is valid, it is necessary to allocate space for the execution of the algorithm as well as for the results. Regarding the execution, due to the high possibility of having a very large tree, it is decided to instead of storing the complete space with all the cases, to dynamically calculate the node's position on the tree and based on that refer to the values that correspond to it to select the data for a particular case. In other words, if a node is the child of the first node of the second level (the first level contains the original case) and is in the second position on the third level, the values that will be used for this experiment are the first option for the first keyword and the second option for the second keyword.

To be able to find the node's position, an array which holds the children of the node in question is allocated. The size of this array is given by the maximum number of values from all the optimization keywords, as each value corresponds to a child. The other execution related element is an array that contains the path to the node. In this case, size is given by the number of groups, as it is the number of levels on the tree. As for the results arrays, three arrays are allocated with a size corresponding to the total number of nodes in the tree, therefore being able to collect all the results. The first of the arrays stores a value that indicates if a result is valid or not based on if it is under the threshold. The other two are to store the cost and the escapes for each of the cases that are run during the optimization.

Execution of the Test Flow Optimization environment

The functions described in this section are in charge of executing the optimization. All information is already stored in memory, so these functions are implemented to correctly select the data to be used during the exploration of the different cases.

Function execute_tf_opt

This function is called by the *main* function. Its objective is to organize the execution of the algorithm. It starts by checking if the optimization is performed based on lower escapes or cost, this does not affect greatly the execution, it just tells the algorithm which of the values to use for guiding the tree exploration.

To explore the tree, a while loop is implemented, with the condition of continue executing the loop if the current node is not the cheapest leaf node. The first node to be explored is node 0 which represents the original case. From there the nodes with the lowest cost or escape value will be explored until finding the solution. This loop can be seen in Figure 5.2.

The loop works by first calling *get_children*, after this function is done, the current node's level is compared against the number of levels to determine if it is a valid leaf node. The validity is given by if the node's cost or escapes is less than the specified threshold. If it is valid, a flag is modified to signal this and stop the execution of the loop.

For the cases where the current node is not the solution, the program continues by calling *update_arrays*, which reruns the cost calculation and updates the results arrays. The last step is performed by *find_min*, which inspects the results arrays to find the node with the lowest cost or escapes value, depending on the optimization target and selects it as the next node to be explored.

Function get_children

The *get_children* function finds the children of the current node being evaluated. To accomplish this, it calls *find_node*. The *find_node* function works by calculating the first and last node of each of the tree levels and comparing the current node to those values to see if if it is between them.

When the level is found, the *start*, *end* and *level* values are passed back to *get_children*, which then uses this information to calculate based on the number of children for the next level, the first child of the current node and store it in the children array. Then, based on the number of children, the first child's index is incremented and the indices stored in the children array, until the number of children in the array is equal to the number of children for that particular node. The number of children is returned by this function.

Function update_arrays

In this function, the specific values for the current node's experiment are passed to the memory locations used during the cost calculation. It is here where the values specified by the user are matched with a node in the tree and used to make a new cost calculation. This means that through this function, if for example, the current node is node 7 in Figure 5.3, the second value from the first parameter (level 2), the second value from the second parameter (level 3) and the best case value from the third parameter will be passed to the corresponding memory locations for those elements in the test class.

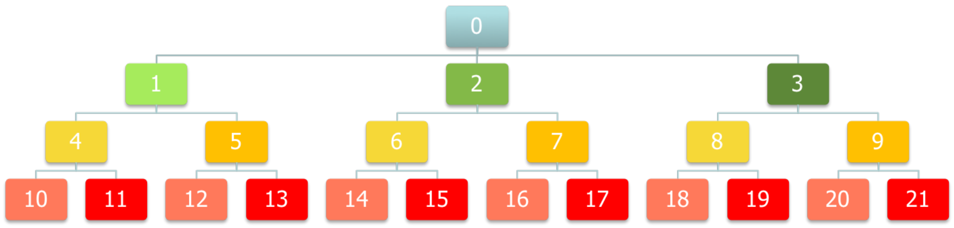


Figure 5.3: Test moments explored in example

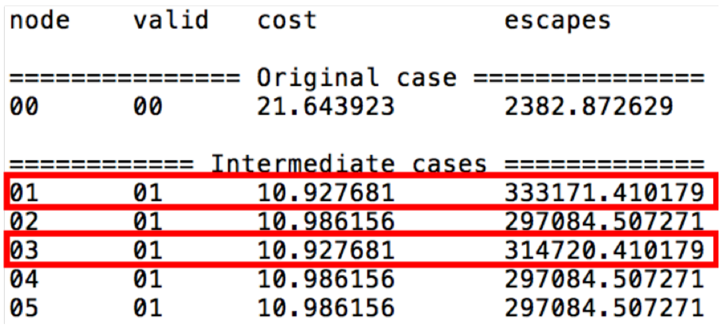
To do so, the *find_path* is used, which through the aid of *find_node* is capable of locating the position of a particular node in the tree, then its relative position on its level is found by subtracting the node's index to the *start* node of the level. After finding the relative position, through modulo operation defined by the number of values at that particular level, the input value to which the node corresponds is found. Following the previous example, level 3 has two options, to identify to which one node 7 corresponds to, first its relative position is found $7-4=3$, then this value is used in $3 mod 2 = 1$ and as the indexing follows C convention, option 1 corresponds to the second value for the parameter in question. This result is stored in the path array at its corresponding position. Then *find_path* moves one level higher and repeats the process for the parent node.

Once the input values are set, the cost is calculated with almost identical cost functions used in the main file. The difference is that the output printing behavior has been modified to prevent overwriting the original case and the results stored in a data collector specific to the optimization.

The last step is to update the results arrays, based on the obtained cost and escapes numbers. Both values are written to their corresponding positions, as well as the valid array, but before setting this one, the results are verified against the threshold to see if they should be considered or if they violate the maximum acceptable value for a leaf node.

Function find_min

The final function called during the execution phase is *find_min*. This function goes through the arrays looking for the lowest value of the primary optimization target, if two nodes have the same value, it then looks into the secondary values for those nodes to find the one with the lowest secondary result and thus select that one. For example, Figure 5.4 shows the results for a case where the optimization is for cost(primary target). The two nodes highlighted in red have the same value, which is the minimum cost, so in this case, node 3 is selected as its secondary target(escapes in this case) is lower than the secondary target of node 1.



```
node      valid     cost                escapes

=============== Original case ===============
00        00        21.643923           2382.872629

============ Intermediate cases =============
01        01        10.927681           333171.410179
02        01        10.986156           297084.507271
03        01        10.927681           314720.410179
04        01        10.986156           297084.507271
05        01        10.986156           297084.507271
```

Figure 5.4: Example results of a test flow optimization case with equal cost

Remaining functions

The remaining functions are in charge of the debug data and the results printing. A summarized version of the optimization results, highlighting the selected solution and the parameters that produce is added to the main results file through *dump_tf_opt_results*. The debug data is included in the main debug file, where *dump_debug_input_tf_opt* reproduces all the input keywords and their respective values. Also in the debug file, a detailed version of the results is printed. The information regarding the best case is included as well as other statistics like the number of explored nodes. Finally, the complete *valid*, *cost* and *escapes* arrays are also present in the debug output, allowing the user to check the results for all the evaluated nodes.

The last function of the **test flow optimization** file, handles the freeing of all the allocated memory, with safeguards to prevent the freeing of unallocated memory, for when the analysis was not performed.

5.1.2. Stacking Order Analysis

Maintaining the same structure as the other classes and optimizations, for the **Stacking Order Analysis** a pair of source and header files was created. For this case, 8 functions are defined and a struct with 9 data elements is utilized. Similarly, a list with the 11 keywords related to the stacking operations are specified, to enable the location in the input file of the stacking operations and further reordering of them.

To perform this optimization a different approach is followed than the one from the **test flow optimization**. Instead of creating the cases and swapping directly into the original case's memory space, copies of the input file are created and modified. During the transcription of the file, the keywords related to the stacking operations are identified and modified to produce the new order, thus producing the same number of input files as cases that are explored. As a file gets created, a system call is made to execute 3D COSTAR with the recently generated file as the target, thus creating a new process.

The advantage of following this approach is that it is more straightforward to rearrange the stacking operations this way, because only the indices must be replaced on a text file. In contrast, if it was performed all in memory, it would have required complex memory manipulations to reorder the data into the new steps. Another advantage is that the used input files remain accessible to the user, so if he would like to explore deeper into one of the possibilities, the input file is already at his reach. As a disadvantage, a penalty is paid due to file accessing and writing operations being significantly slower than operating directly in memory, thus taking a considerable longer time to perform this analysis. Besides the input file and respective debug and results file, a summary file is also created. This file contains the index by which the experiments are identified, the order employed on each file and the cost for that particular experiment, facilitating the analysis of results, by condensing them in one place.

The following sections will explore the functions used to implement this analysis.

Function set_stacking_order_experiment

This is the only function called by *main*. It directs the execution of the analysis, starting by scanning the input file looking for the keyword that specifies the analysis. After confirming the execution, the number of cases is calculated through the permutation of the stacking operations, as described by Equation 5.3. The next step is to determine all the orders and store them in memory, which is done by *determine_orders*, explained in the following section.

$$\text{Number of cases} = (\#\ \text{stackOps})! \tag{5.3}$$

A new folder **stacking_order** is created inside the results folder of the main file, this new folder contains all the generated input files, their results folders and the summary file, which is also initialized at this point. After this, a for loop which iterates until the number of files is met, is used to update the summary file, still in the loop, through *create_new_so_file* the specific input files are created and filled. Also the system call is made to perform the analysis.

Once done with generating all the files and running the tool on them, the memory allocated is freed by calling *free_stacking_order*.

Function determine_orders

The *determine_orders* function provides the framework for calculating all the different stacking orders that will be explored and assigning them to their particular case. It begins by creating a representation of the original case's stacking order and storing it in an array. This array is then passed to the *permute* function, where the process of reordering the operations is made with the aid of *swap* function.

Function permute

The permute function receives the stacking order array and through swapping array elements is able to create the permutations. The *permute* function receives the array and a *start* point and an *end* point. To create all the different possibilities, it uses recursion, where inside the *permute*function a call to itself is done with the intention of moving through the complete array. The *swap* function receives the location of the two array elements it must swap and through pointer swapping, changes the values.

The function works by setting a *start* point and *end* point. During the function's first call, they correspond to the *start* point and *end* point of the array. The first action inside *permute*, is a comparison of those points. It checks if *start* and *end* are equal. If they are, the array has been completely traversed, therefore its order is stored.

When the previous condition is not met, a for loop is implemented. Inside this loop is where the recursive call occurs. The index i of the loop is initialized with the *start* value passed to it. The first step is to swap *start* with i, which in this case are the same position, thus no change occurs. After this, *permute* is called, recreating this process. However, the parameters passed to it have been modified. A new *start* point, corresponding to *start*+1 is used in the recursive *permute* calls. The new *start* point inside the recursive call moves the elements on which the swapping functions are operating.

This process repeats itself until the evaluation is successful, where it backtracks and increments the higher level's loop index by 1. After the increment, the loop is executed again when the new i is still under the loop's limit. When the end condition of the loop is met,the current execution of the recursive call is finished, falling back into the higher level from which it was called. This process keeps occurring until the initial *permute* reaches the end of its loop, thus all the possible permutations having been found. Figure 5.5 exemplifies this process. An array representing the stacking order for a six stacking operations process is presented. By going through the permute operation, the different orders are obtained.

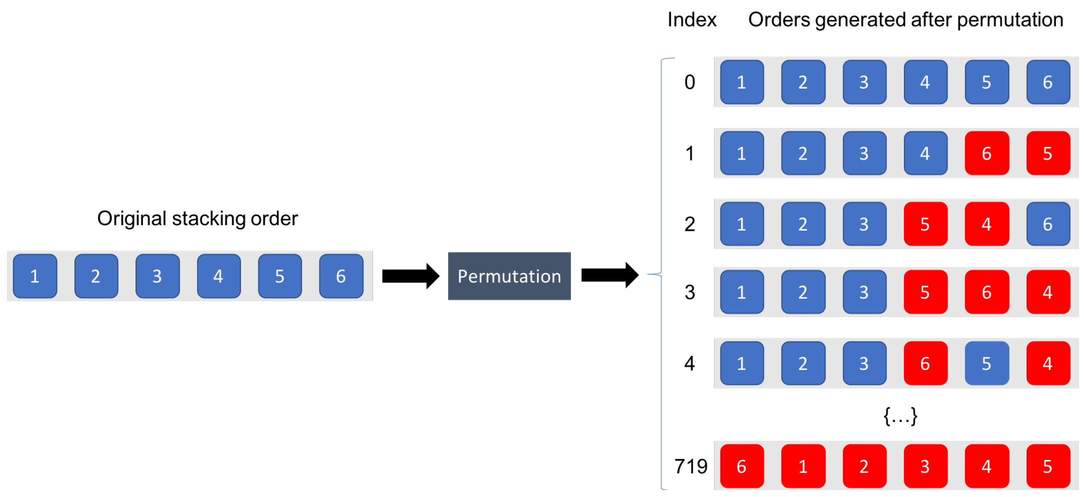


Figure 5.5: Stacking order determination

Function create_new_so_file

The function begins by creating the new input file, then line by line, the original input file is read and copied into the new file. During the copying process the contents are also analyzed, with the intention of finding the activation keywords for the sensitivity analysis, test flow optimization or stacking order, as well as the experiment name. The reason for this is that for the stacking order generated input files, all other analysis must be deactivated to prevent infinite loops or unnecessary simulations. The experiment name is modified to include the file number at the end, to properly identify each file.

Once the copy process is done, a for loop iterates over the stacking operation commands, to find their position in the file. This process is done for every keyword related to a stacking operation, so if there were three stacking operations, every keyword is searched for stacking operation 1, 2 and 3. The function in charge of locating the keywords position can be found in the read_io file, although it was created for this case. Still inside the for loop, the *replace* function is called. Inside *replace*, for each stacking operation, the previously found locations are passed to another read_io function *replace_word_position*, where the index of the operation is swapped from the original order to the one pertaining to this particular case. For example, if stacking operation 2 from the original case will be swapped with 1, it is at this point where the keywords ending in *stackOp[2]* are replaced with *stackOp[1]*. In a few cases, the stacking operations are specified on a column base, instead of per row, for this cases, another function separates each column's value and then swaps them, thus covering all the stacking operation keywords. After this, the input file is closed and the system call to 3D COSTAR is made with a special command line argument to indicate that the execution is related to the **stacking order analysis**. A child process is generated in charge of executing the cost analysis to the recently created input file.

To collect the results in the summary file, the obtained cost must be passed, but as the execution of 3D COSTAR is done on a separate process, there is no direct way to access the value. To solve this, the previously mentioned command line argument, will signal for the creation of a temporary file during the child process' read_io initialization. The temporary file will only contain the cost obtained after running the experiment. By doing it this way, time is saved by not having to open the regular output file and searching through all the information to find the desired value, instead, the program only has to open the temp file and copy the only value that exists. After the value has been copied, the temporary file is deleted and the folder containing the results for the particular case are moved into the **stacking_order** folder.

Function dump_stacking_order

This function's task is to inform the user about the experiment's execution. When the analysis is requested by the user, it prints into the main output file the number of cases that will be created. As previously mentioned, the actual results will be available through each case's results folder and the summary file.

5.1.3. TSV Manufacturing Cost

As this last extension is not an algorithm or an operation, but a specific parameter related to the manufacturing process, the way of implementing this differs from the previously discussed cases. In this occasion, no files are created, but changes are made to the manufacturing class.

Due to the complexity and number of parameters involved during the manufacturing stage, the main manufacturing data is divided into small structs based on the part of the manufacturing process that they represent. For example, there is one struct holding the costs of the manufacturing process, one with the yields of the wafer processing part, another one with the yields related to the stacking operations and like this, others until covering the whole manufacturing process. To incorporate the cost of manufacturing the TSVs, a new variable of type double (*cost_interconnect*) is declared inside the costs struct. This variable will hold the value specified by the keyword *M_cost_interconnect*, which represents the cost of creating the TSVs on a per wafer basis, for each of the dies being manufactured. This values are read at the initialization stage of the manufacturing section.

Throughout the execution of the manufacturing modeling, the cost variables are not utilized until the end, as the intermediate steps in this process are related to the physical aspect of the manufacturing phase, thus more concerned with the yields that characterize the modeled processes. It is at the end of the manufacturing modeling, where the costs get involved. They are used in the calculations to obtain the final results, which are reported in the output and debug files.

As the parameter specifies the cost per wafer, it is included in the **Manufacturing cost per die** calculation seen in the main output file, Equation 5.4. The **TSV Manufacturing Cost** is also used to calculate results included in the debug file as the **Manufacturing cost per die of the interconnect** and **Manufacturing cost per die in final shipped 3D SICs**, which differs from the **Manufacturing cost per die**, as **Manufacturing cost per die in final shipped 3D SICs** is based only on the stacks that were deemed good after all the testing phases, whereas **Manufacturing cost per die** is more like a gross figure, which only considers the manufactured units.

$$\text{M\_cost\_per\_die} = \frac{\text{M\_cost\_wafer} + \text{M\_cost\_interconnect}}{\text{M\_diesPerWafer}} \tag{5.4}$$

$$\text{M\_cost\_per\_die\_interconnect} = \frac{\text{M\_cost\_interconnect}}{\text{M\_diesPerWafer}} \tag{5.5}$$

$$\text{M\_cost\_per\_die\_shipped} = \frac{\frac{\text{M\_cost\_wafer} + \text{M\_cost\_interconnect}}{\text{M\_diesPerWafer}} + \text{test\_cost\_die\_prebond}}{\text{ratio\_dies\_in\_stack}} \tag{5.6}$$

5.2. Case studies

In this section, three case studies will be presented, one for each stacking architecture (2.5D, 3D and 5.5D). First their input parameters will be specified and after that, the results obtained after applying the 3D COSTAR optimizations to the cases will be shown and discussed.

Table 5.1: Parameters for FPGA 2.5D case

Parameter	Interposer	FPGA dies
Wafer costs ($)	700	3000
Effective wafer radius (mm)	147	147
Die Area (mm^2)	460	150
Defect density (mm^{-2})	0.001	0.005
Die yield (%)	72.17	63.24
Pre-bond FC (%)	0	99
Pre_bond test cost ($)	0	1.50
Stacking cost ($)	0	0.05
Stacking die yield (%)	99.5	99
Stacking Interconnect yield (%)	-	99
Mid/post-bond FC (%)	0	99
Mid/post-bond cost ($)	0	1.50
Final FC die (%)	100	100
Final die test cost ($)	0.05	1.50

For the stacking order analysis, each case will be explored in the totality of the stacking order possibilities, reporting the cheapest option. In the case of the test flow optimization, the analysis will be performed both for cost as well as for escapes and for different thresholds, to find the cheapest solution at each point. The costs are considered in dollars and the escapes figures are in DPPM. The tree will be constructed by evaluating all the test moments specified in the original case, based on a binary decision of applying the specified test or not. Each pre-bond test will be evaluated independently, thus if there are four dies, the first four levels of the tree will correspond to those dies, where for each die, two options will be available, to test with the FC and cost mentioned in the original case, or to not test. The tests applied during each stacking operation and the packaging test will be grouped based on the same operation, so the decision will be to apply all of the tests specified during each particular stacking operation, or to not apply any of the tests during that stacking operation. In other words, if during stacking operation 2, dies 1 and 2 had each a test with FC 97% and cost $1, the decision will be if both of these tests will be applied or not at that particular moment.

5.2.1. Case Study 1, FPGA 2.5D stack

The FPGA case study is based on a 2.5D stack composed by four dies. One die is the interposer which serves as the base were the three identical FPGA dies will be stacked on a F2F manner. As the extensions are related to manufacturing and testing, only these two classes will incur in costs, thus allowing an easier comparison between the different experiments.

For the wafer processing phase, a standard 300mm diameter wafer with effective radius of 147mm is considered. As the interposer is a passive component, its cost is lower corresponding to $700. For the FPGAs, the manufacturing cost per wafer is $3000. The dimensions of the interposer dies are 21.4476mm per side, resulting in an area $A = 460$ mm^2. As for the FPGAs, the dimensions are 12.2474mm for an area $A = 150$ mm^2. The defect density for the interposer is $d_0 = 0.001$ defects/mm^2 and for the FPGAs $d_0 = 0.005$ defects/mm^2 and the defect clustering parameter $\alpha = 0.5$ for both cases.

As there are four dies, three stacking operations will occur. Due to the stack being of the 2.5D kind, during all three operations a die will be stacked directly on top of the interposer, therefore, the stacking yield will be of 99.5% for the interposer and 99% for the die being stacked. The cost of performing a stacking operation will be $0.05.

As for the testing, the interposer will only be tested at the packaging test moment, with a 100% FC and a cost of $0.05, due to the fact that to access the FPGAs for testing, it is necessary to go through the FPGA, thus a defective interposer can be indirectly detected during the previous moments. For the FPGAs, a pre-bond test with cost $1.50 and FC 99% will be applied. During the stacking operations, the die stacked on each operation will be tested with the same FC and cost. Finally, at the final test or packaging test, the FPGAs will be retested with $1.50 and FC 100%.

Results Analysis

The stacking order analysis results, can be seen in Table 5.2 and Figure 5.6. From the different possibilities, the cheapest solution is denoted by index 1, which consists on swapping the second and third stacking operations. It constitutes a difference of only 0.67%, which is expected as the only differing parameters between the different cases are the application of tests in parallel.

Table 5.2: FPGA case stacking order results

Index	Order	Cost
0	1 2 3	59.29
1	1 3 2	58.90
2	2 1 3	59.06
3	2 3 1	59.06
4	3 2 1	60.52
5	3 1 2	60.12

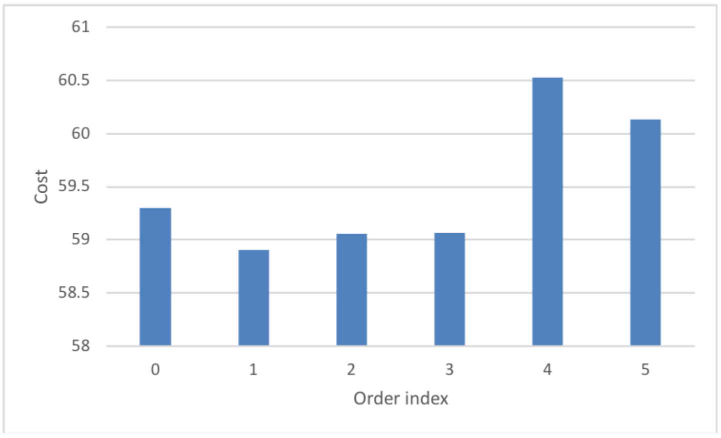


Figure 5.6: Cost of different stacking orders FPGA case

For the test flow optimization, Tables 5.3 and 5.4 contain the results of performing the optimizations targeting lower cost and lower escapes respectively. The cost of the original case is $59.29 and there are 0 escapes. All the best nodes in both optimizations achieve a lower cost, but in many cases a there is trade off in terms of the quality of the test.

Table 5.3: Results of optimizing for lower cost in FPGA case

Threshold	Node	Cost	Escapes	Cases explored
No threshold	255	28.27	825491.17	16
100000	495	51.45	44182.83	240
75000	495	51.45	44182.83	240
50000	495	51.45	44182.83	240
40000	503	53.09	29822.65	260
30000	503	53.09	29822.65	260
20000	501	55.27	15222.09	292
10000	496	55.87	0.00	294

The binary tree for this case, is based on 28 input parameters grouped into 8 groups , in addition of the original case, making it a 9 level tree. This tree structure has a total of 511 nodes and 255 leaf nodes. By analyzing the results from Table 5.3, it is evident the effect of tightening the threshold. As fewer escapes are tolerated, the cost goes up. This inverse relation is explained because by lowering the accepted escape value, more tests must be applied, which is confirmed by the different test flows selected to obtain the previous results. As the testing strategy gets more strict, fewer units pass the tests so it is necessary to manufacture more dies and stacks to comply with the required number of produced units, so the process becomes more expensive. Figure 5.7, presents a visual representation of these results, where it is possible to clearly observe as the results move in opposite directions in terms of cost and escapes.

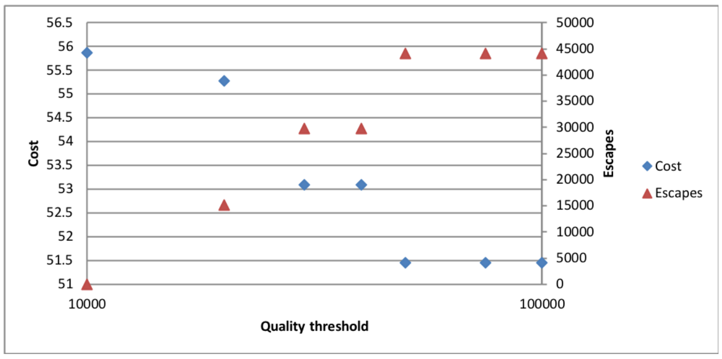


Figure 5.7: Cost and escapes for optimization for lower cost in FPGA case

Figure 5.8 contains the results in terms of cases explored for each threshold. For the higher threshold cases, the result is favorable in comparison to an exhaustive search case, which would explore a number of cases equal to the total number of leafs. As the threshold become more strict, it is worse than the exhaustive search due to the high variability of the input conditions, which creates a tree with very extreme results from nodes close to each other.

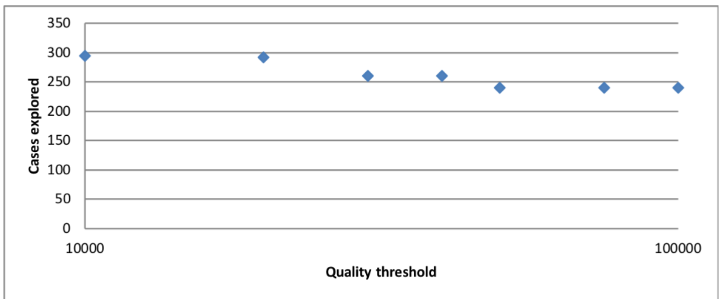


Figure 5.8: Number of cases explored to find cost solution in FPGA case

In the case of the optimization targeting escapes, presented in Table 5.4 and Figures 5.9, 5.10, a similar inverse relation is observed. The same trend occurs in this case, but due to the optimization being done by finding the lowest escapes for a certain cost limit, the order of the results is reversed. Although the cost and escapes behave in a similar way, the cases explored to find the best solution gets worse, having to explore the complete tree. The reason for this is because the test or no test binary decision, produces very extreme results. Since the tree is a binary tree, leaf nodes are organized in pairs, obtained by following the same test flow up until the last test moment. On the last test moment one of the nodes follows the no test route while the other does get tested and this last test is the difference between catching all the defects or not.

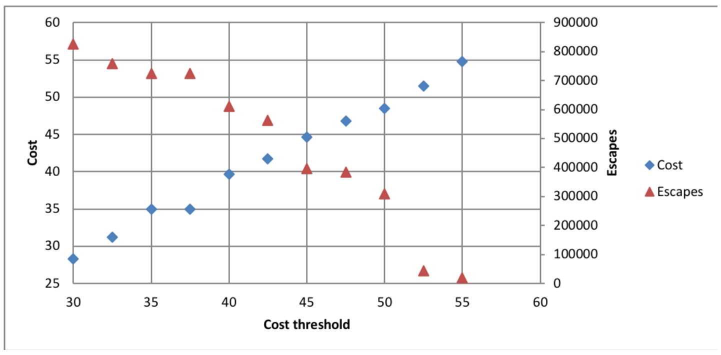


Figure 5.9: Cost and escapes for optimization for lower escapes in FPGA case

Table 5.4: Results of optimizing for lower escapes in FPGA case

Threshold	Node	Cost	Escapes	Cases explored
No threshold	496	55.87	0.00	38
55	507	54.75	20121.36	510
52.5	495	51.45	44182.83	510
50	367	48.51	310198.21	510
47.5	433	46.80	383336.60	510
45	431	44.64	395486.70	510
42.5	303	41.70	563730.00	510
40	455	39.66	611927.21	510
37.5	271	34.95	724077.97	510
35	271	34.95	724077.97	510
32.5	383	31.21	758193.53	510
30	255	28.27	825491.17	510

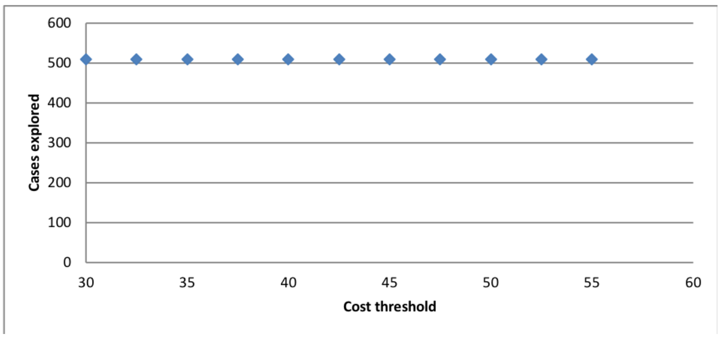


Figure 5.10: Number of cases explored to find escapes solution in FPGA case

5.2.2. Case Study 2, Memory 3D Stack

The second case study explores a 3D stack also composed by four dies. In this case, all four dies are identical memory dies stacked on a F2B manner.

For the wafer processing phase, a standard 300mm diameter wafer with effective radius of 147mm is considered. The manufacturing cost per wafer is \$2279.30. The dimensions of the dies are 7.0711mm per side, resulting in an area $A = 50$ mm^2. The defect density is $d_0 = 0.005$ defects/mm^2 and the defect clustering parameter $\alpha = 0.5$.

As in the FPGA case, three stacking operations will occur. In this case, the stacking operations will be performed on a die to die basis, initially stacking die 2 on top of die 1, followed by die 3 on top of die 2 and die 4 on die 3. Only the die being stacked and the one where it will be stacked are affected by the process, so the stacking yield will be of 99% for the two dies directly involved and their interconnects and 100% for the others in the stack. The cost of performing a stacking operation will be \$0.10.

As for the testing, all dies will be tested during the pre-bond phase with FC = 99.6% and a cost of \$0.05. During the first two stacking operations, no tests will be performed. After the third operation and packaging, all the dies in the stack will be tested with 100% FC and \$0.10 cost. Similarly, interconnect tests are applied in these two moments with a FC of 100% FC and a cost of \$0.05.

Results Analysis

The result obtained from the original case is a cost of \$15.09 an an escapes figure of 0 DPPM, as the final test after packaging has a 100% FC. Table 5.6, contains the stacking analysis results for the memory case. Through Figure 5.11, a visual representation is presented, where it is easy to observe how the stacking order denoted by index 3 produces the cheapest solution. The reason for this is that by moving the test moment with 100% FC to the middle of the process, an earlier detection of the defective stacks is accomplished.

In the original case, the test is performed after all three operations have been done, but if a defect was introduced after the first stacking operation, it would remain undetected until the end and two more dies would be wasted on a bad stack; the test would have been applied too late. Moving the test to the middle of

Table 5.5: Parameters for Memory 3D case

Parameter	Memory dies
Wafer costs ($)	2279.30
Effective wafer radius (mm)	147
Die Area (mm^2)	50
Defect density (mm^-2)	0.005
Die yield (%)	81.65
Pre-bond FC (%)	99.6
Pre_bond test cost ($)	0.50
Stacking cost ($)	0.10
Stacking die yield (%)	99
Stacking Interconnect yield (%)	99
StackOp 3/post-bond FC (%)	100
StackOp 3/post-bond cost ($)	0.10
Final FC die (%)	100
Final die test cost ($)	0.10

Table 5.6: Memory case stacking order results

Index	Order	Cost
0	1 2 3	15.09
1	1 3 2	14.89
2	2 1 3	14.96
3	2 3 1	14.74
4	3 2 1	15.10
5	3 1 2	15.15

the stacking process, permits the filtering of those defective units at a time where less resources have been dedicated to them.

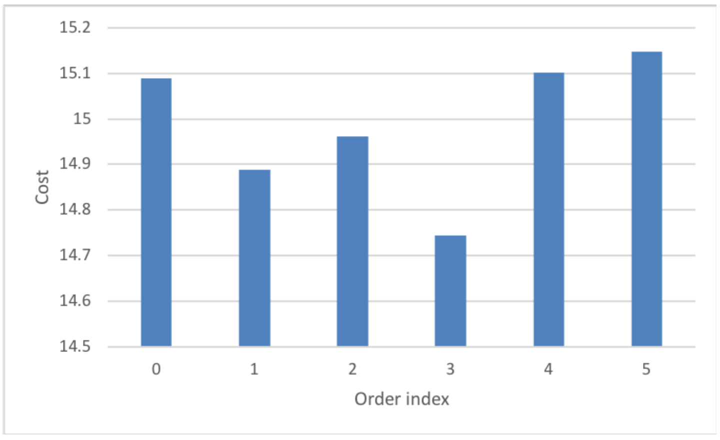


Figure 5.11: Cost of different stacking orders memory case

For the test flow optimization, a similar behavior is observed as in the FPGA case, where there is an inverse relation between cost and escapes as the threshold is modified.

Eight different test moments are identified in this case, creating a similar binary tree consisting of 9 levels, with the difference that 36 input parameters exist in this case, but are also grouped into 8 groups. The tree has 511 total nodes and 255 leaf nodes.

As mentioned previously, a similar trend in terms of how the cost and escapes move is recognized in this case, which can be observed in Table 5.7 and Figures 5.12 and 5.14. The reason for this remains the same, as the testing strategy becomes more strict, fewer units pass the tests so it is necessary to manufacture more dies and stacks to comply with the required number of produced units, so the process becomes more expensive.

Table 5.7: Results of optimizing for lower cost in memory case

Threshold	Node	Cost	Escapes	Cases explored
No threshold	255	8.66	598052.43	510
100000	495	12.70	98028.00	188
75000	497	13.20	70418.50	190
50000	503	13.86	39403.99	196
40000	503	13.86	39403.99	196
30000	499	14.07	29701.00	200
20000	505	14.39	10000.00	202
10000	501	14.62	0.00	208

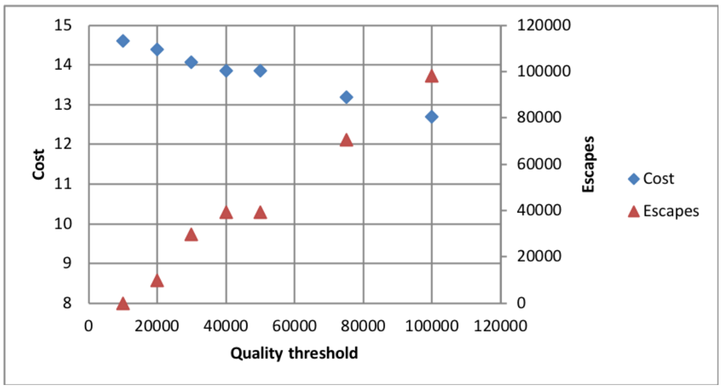


Figure 5.12: Cost and escapes for optimization for lower cost in memory case

Figure 5.13 shows the number of nodes explored to obtain the results for the lowest cost case. In this opportunity, the results are lower, signifying that the algorithm was able to find the solution in a more efficient way.

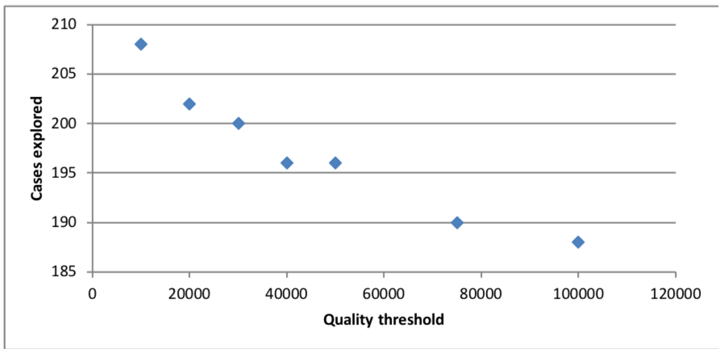


Figure 5.13: Number of cases explored to find cost solution in memory case

In the case of the optimization targeting escapes, presented in Table 5.8 and Figure 5.15, the number of cases is very close to the total number of nodes in the tree. As in the FPGA case, the extremeness of the conditions observed in the tree, produces drastically different escape results, therefore the algorithm is not capable of properly optimizing the solution, which becomes more like a brute force approach.

5.2.3. Case Study 3, 5.5D Stack

The last case consists of a more complex device than in the previous cases. It is made up of seven dies arranged in a 5.5D way. Due to the steps required to manufacture it, it utilizes two carrier wafers during intermediate steps in the manufacturing process to, provide support to the stack when not all of the parts are yet in place.

From the seven dies, one is a logic die, which occupies the central position, there are two memory dies, which will be placed one on the right side and the other on the left side of the logic die. Two silicon bridges

Table 5.8: Results of optimizing for lower escapes in memory case

Threshold	Node	Cost	Escapes	Cases explored
No threshold	501	14.62	0.00	38
14	503	13.86	39403.99	446
13.5	499	13.20	70418.50	450
13	495	12.70	98028.00	450
12.5	371	12.16	240493.39	462
12	367	11.69	263051.49	466
11.5	339	11.12	379451.62	484
11	303	10.68	397882.52	490
10.5	291	10.08	492986.25	498
10	271	9.67	508045.06	506
9.5	259	9.04	585748.75	510
9	255	8.66	598052.43	510

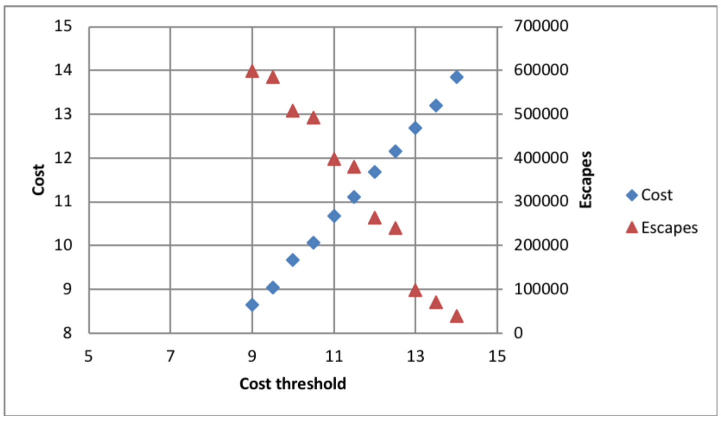


Figure 5.14: Cost and escapes for optimization for lower escapes in memory case

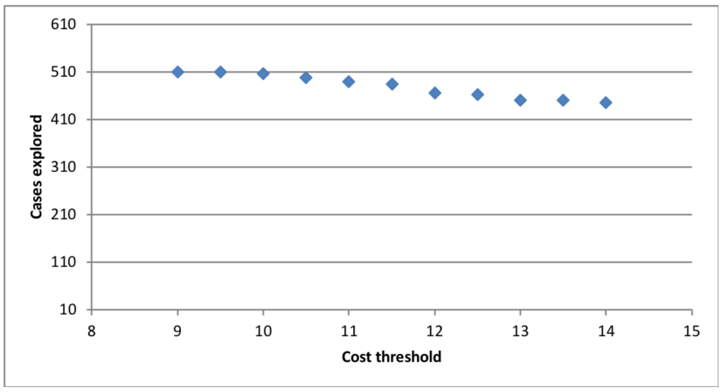


Figure 5.15: Number of cases explored to find escapes solution in memory case

are used to create the base of the connection between the memories and the logic die and finally, two TPV (through package via) dies provide access.

Although there are nine dies used, for the modeling of the manufacturing process an additional dummy device is defined, which permits a more accurate representation of the stacking process, therefore the 3D COSTAR case consists of ten dies and nine stacking operations.

Results Analysis

The results for the original case are a cost of \$21.64 with an escapes number of 2382.87 PPM. Due to the number of stacking operations, the total number of stacking orders is 9!=362880. The best case based on the stacking order analysis produces a cost of $18.16, which is 16% less than the cost from the original order. It is achieved by ordering the stacking operations in the order found on stacking order 36044, which corresponds to 1,9,4,3,6,2,8,7,5. Due to the large number of results, the table with the stacking order analysis results has been omitted.

For the test flow optimization 108 parameters have been used organized in 11 groups, for a total of 4095 nodes and 2048 leaf nodes.

Table 5.9: Results of optimizing for lower cost in 5.5D case

Threshold	Node	Cost	Escapes	Cases explored
No threshold	2047	10.78	339816.40	22
300000	3583	10.89	266428.36	78
200000	2815	11.87	190813.32	214
100000	3615	12.68	88846.45	418
75000	3599	13.30	27675.02	498
50000	3599	13.30	27675.02	498
40000	3599	13.30	27675.02	498
30000	3599	13.30	27675.02	498
20000	3791	14.74	16265.05	1462
10000	3600	15.83	2382.87	2590
5000	3600	15.83	2382.87	2590
2500	3600	15.83	2382.87	2590

The results for the lower cost optimization are shown in Table 5.9 and Figures 5.16, 5.17. They display the similar trends to the previous cases, following an inverse relation between the cost and escapes as the threshold becomes smaller and the number of cases explored increases. Similarly to the Memory case from Section 5.2.2, the cases explored take advantage of the optimization for the looser scenarios, but as the threshold becomes tighter and gets closer to the original case's escapes, it becomes more difficult to find a solution that meets the requirements.

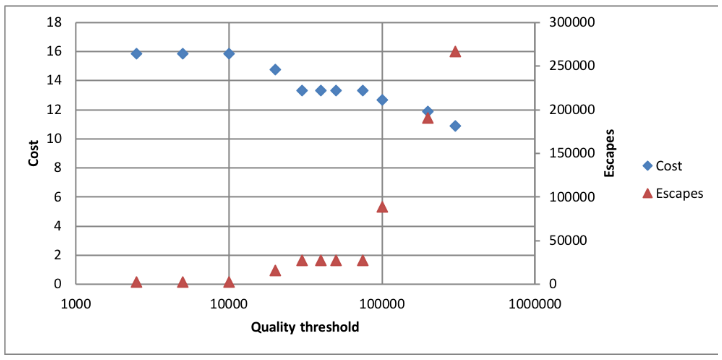


Figure 5.16: Cost and escapes for optimization for lower cost in 5.5D case

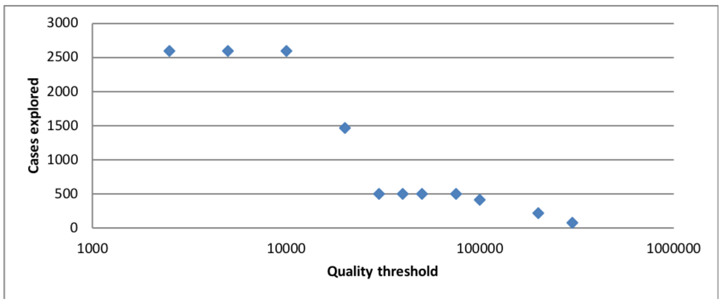


Figure 5.17: Number of cases explored to find escapes solution in 5.5D case

When evaluating the optimization for lower escapes, the same inverse relation is identified across Table 5.10 and is also visible in Figure 5.18. This is expected even though the parameters of the case differ greatly to the other two cases, but the options presented during the test flow optimization follow the same binary decision structure.

Table 5.10: Results of optimizing for lower escapes in 5.5D case

Threshold	Node	Cost	Escapes	Cases explored
No threshold	3600	15.83	2382.87	106
15.5	3791	14.74	16265.05	3072
15	3791	14.74	16265.05	3072
14.5	3919	14.25	21986.67	3072
14	3855	13.53	27675.02	4094
13.5	3599	13.30	27675.02	4094
13	3871	12.95	88846.45	4094
12.5	3839	11.92	147022.40	4094
12	3839	11.92	147022.40	4094
11.5	3583	10.89	266428.36	4094
11	3583	10.89	266428.36	4094

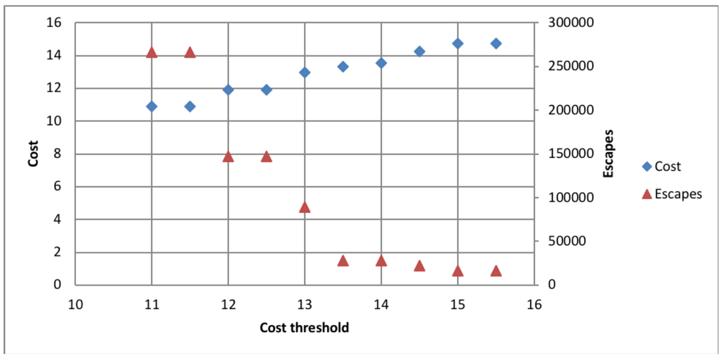


Figure 5.18: Cost and escapes for optimization for lower cost in 5.5D case

As for the cases explored to find the solutions, the trend does not reverse for the 5.5D case, requiring the exploration of more nodes than if performing an exhaustive search analysis.

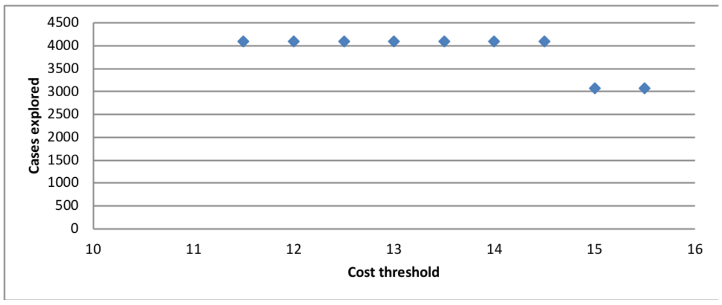


Figure 5.19: Number of cases explored to find escapes solution in 5.5D case

6 Conclusion

This chapter presents a summary of the activities carried out throughout the thesis, it also states future work to be done related to the topics discussed. Section 7.1 presents a summary of the activities. Section 7.2 states the future work.

6.1. Summary

Chapter 1 detailed the process by which the semiconductor industry arrived at the stacked IC technology. Motivated by reaching the limits imposed by the physical properties of the materials, the industry had to find an alternative way to keep improving device performance. Also in this chapter, as the transition to stacked ICs is made, variations in the process have a large impact on cost. Existing models used during the planning stages must be adapted to work with this new scenario, highlighting the need for updated models and tools to provide accurate results describing the stacked IC process.

Chapter 2 described the different kinds of stacked ICs as well as the manufacturing process by which stacking of dies is achieved. It presented the challenges and opportunities, displaying the potential for this technology, without ignoring the difficulties still ahead.

Chapter 3 introduced the cost modeling concept and explored how it has been applied to 3D SIC. Many efforts have been made by researchers around the world, but the existing solutions can still be improved. The 3D COSTAR model and tool are described, demonstrating how it is capable of performing the complex tasks involved in the 3D SIC process modeling. The limitations of the tool are discussed, laying the ground to the extensions, as they are proposed as alternatives to broaden the capacities of the tool.

Chapter 4 presents the proposed methods by which three of 3D COSTAR limitations will be addressed. In this chapter, the selected limitations to be solved are explored in a deeper manner, demonstrating the reasons for focusing on those specific cases. Each limitation is transformed into an extension, which has as its objective to improve the tool. These extensions are discussed throughout the chapter, explaining the theory behind them and the impact that they will have.

Chapter 5 delves into the extensions, exploring the details regarding their implementation, discussing the structure and functionality of the code. Also in this chapter, three case studies are presented, each one addressing a different type of stacked IC. The results for those cases are discussed based on a typical usage scenario for the tool, demonstrating how it is capable of finding the best test flow based on the users needs, as well as analyzing the stacking order to find a different possible order that entails a lower cost.

6.2. Future Work

In this section, several recommendations are suggested regarding future work related to the topics discussed in this thesis.

- Adapt the code to take advantage of multi-core processor and GPU parallelism, calculating multiple cases at the same time and shortening the run time.
- Include in the model the possibility of identifying in the stacking operations if there is access to perform the tests, as in some cases, I/O infrastructure might be blocked, creating scenarios during the stacking order analysis where it is not possible to perform the tests.

- Based on the tests specified in the input file, automate the generation of the input keywords for a binary decision tree in the test flow optimization, recognizing the tests being performed and ignoring all the elements which are not being tested. As showcased during the case studies in Section 5.2, evaluating if the tests specified in the input file should be performed or not can shed valuable information regarding lowering the cost of the process, while still manufacturing at an acceptable escapes rate.

Bibliography

[1] Calibre for 3d-ic sign off. URL `https://www.mentor.com/solutions/3d-ic-design/calibre`.

[2] Update on jedec thermal standards. URL `https://www.electronics-cooling.com/2012/09/update-on-jedec-thermal-standards/`.

[3] Through-silicon via, Feb 2018. URL `https://en.wikipedia.org/wiki/Through-silicon_via`.

[4] K. Acharya, K. Chang, Bon Woong Ku, S. Panth, S. Sinha, B. Cline, G. Yeric, and S. K. Lim. Monolithic 3d IC design: Power, performance, and area impact at 7nm. In *2016 17th International Symposium on Quality Electronic Design (ISQED)*, pages 41–48, March 2016. doi: 10.1109/ISQED.2016.7479174.

[5] Mukesh Agrawal and Krishnendu Chakrabarty. Test-cost optimization and test-flow selection for 3d-stacked ICs. In *VLSI Test Symposium (VTS), 2013 IEEE 31st*, pages 1–6. IEEE, 2013. URL `http://ieeexplore.ieee.org/abstract/document/6548941/`.

[6] Mukesh Agrawal and Krishnendu Chakrabarty. Test-Cost Modeling and Optimal Test-Flow Selection of 3-D-Stacked ICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34 (9):1523–1536, September 2015. ISSN 0278-0070, 1937-4151. doi: 10.1109/TCAD.2015.2419227. URL `http://ieeexplore.ieee.org/document/7096976/`.

[7] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications*. Pearson, Harlow, pearson new internat. ed edition, 2014. ISBN 978-1-292-04270-1. OCLC: 881523756.

[8] R. Anigundi, H. Sun, J. Q. Lu, K. Rose, and T. Zhang. Architecture design exploration of three-dimensional (3d) integrated DRAM. In *2009 10th International Symposium on Quality Electronic Design*, pages 86–90, March 2009. doi: 10.1109/ISQED.2009.4810274.

[9] K. Athikulwongse, A. Chakraborty, J. S. Yang, D. Z. Pan, and S. K. Lim. Stress-driven 3d-IC placement with TSV keep-out zone and regularity study. In *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 669–674, November 2010. doi: 10.1109/ICCAD.2010.5654245.

[10] C. E. Bauer and H. J. Neuhaus. Advanced Opportunity Cost Analysis of 3d Packaging Technologies. In *2008 10th Electronics Packaging Technology Conference*, pages 338–343, December 2008. doi: 10.1109/EPTC.2008.4763457.

[11] C. E. Bauer and H. J. Neuhaus. 3d device integration. In *2009 11th Electronics Packaging Technology Conference*, pages 427–430, December 2009. doi: 10.1109/EPTC.2009.5416508.

[12] Mohamed Benabdeladhim, Belgacem Hamdi, and Aymen Fradi. Based IBIST auto-parallel reconfiguration of TSV defect in 3d-IC. pages 1–6. IEEE, March 2015. ISBN 978-1-4799-8172-4. doi: 10.1109/WSWAN.2015.7210312. URL `http://ieeexplore.ieee.org/document/7210312/`.

[13] Eric Beyne and Ingrid Dewolf. Failure analysis for 3d tsv systems, 2013. URL `https://pdfs.semanticscholar.org/presentation/64d0/58a85a53f177a4d2bf8eb1483e2e74b81911.pdf`.

[14] Michael L. Bushnell and Vishwani D. Agrawal. *Essentials of electronic testing for digital, memory, and mixed-signal VLSI circuits*. Number 17 in Frontiers in electronic testing. Kluwer Academic, Boston, 2000. ISBN 978-0-7923-7991-1.

[15] K. Chakrabarty, S. Deutsch, H. Thapliyal, and F. Ye. TSV defects and TSV-induced circuit failures: The third dimension in test and design-for-test. In *2012 IEEE International Reliability Physics Symposium (IRPS)*, pages 5F.1.1–5F.1.12, April 2012. doi: 10.1109/IRPS.2012.6241859.

[16] Ying-Wen Chou, Po-Yuan Chen, Mincent Lee, and Cheng-Wen Wu. Cost modeling and analysis for interposer-based three-dimensional IC. In *VLSI Test Symposium (VTS), 2012 IEEE 30th*, pages 108–113. IEEE, 2012. URL `http://ieeexplore.ieee.org/abstract/document/6231088/`.

[17] W. R. Davis, J. Wilson, S. Mick, J. Xu, H. Hua, C. Mineo, A. M. Sule, M. Steer, and P. D. Franzon. Demystifying 3d ICs: the pros and cons of going vertical. *IEEE Design Test of Computers*, 22(6):498–510, November 2005. ISSN 0740-7475. doi: 10.1109/MDT.2005.136.

[18] Joeri De Vos, Vladimir Cherman, Mikael Detalle, Teng Wang, Abdellah Salahouelhadj, Robert Daily, Geert Van der Plas, and Eric Beyne. Comparative study of 3d stacked IC and 3d interposer integration: Processing and assembly challenges. In *3D Systems Integration Conference (3DIC), 2014 International*, pages 1–7. IEEE, 2014. URL `http://ieeexplore.ieee.org/abstract/document/7152146/`.

[19] Ji Fan and Chuan Seng. Low temperature wafer-level metal thermo-compression bonding technology for 3d integration. *Metallurgy - Advances in Materials and Processes*, 2012. doi: 10.5772/48216.

[20] Philip Garrou, Christopher Bower, and Peter Ramm, editors. *Handbook of 3D integration: volumes 1 and 2 ; technology and applications of 3D integrated circuits*. Wiley-VCH-Verl, Weinheim, 2012. ISBN 978-3-527-33265-6. OCLC: 819510958.

[21] Daniel Gitlin, Maud Vinet, and Fabien Clermidy. Cost model for monolithic 3d integrated circuits. In *SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S), 2016 IEEE*, pages 1–2. IEEE, 2016. URL `http://ieeexplore.ieee.org/abstract/document/7804408/`.

[22] Armin Gruenewald, Michael Wahl, and Rainer Brueck. Cost modeling and analysis for the design, manufacturing and test of 3d-ICs. In *3D Systems Integration Conference (3DIC), 2015 International*, pages TS8–32. IEEE, 2015. URL `http://ieeexplore.ieee.org/abstract/document/7334603/`.

[23] GWOL. Silicon photonics. URL `https://www.colorado.edu/engineering/gwol/SiliconPhotonics.html`.

[24] D. Henry, F. Jacquet, M. Neyret, X. Baillin, T. Enot, V. Lapras, C. Brunet-Manquat, J. Charbonnier, B. Aventurier, and N. Sillon. Through silicon vias technology for CMOS image sensors packaging. In *2008 58th Electronic Components and Technology Conference*, pages 556–562, May 2008. doi: 10.1109/ECTC.2008.4550028.

[25] C. S. Hou, Y. X. Chen, J. F. Li, C. Y. Lo, D. M. Kwai, and Y. F. Chou. A built-in self-repair scheme for DRAMs with spare rows, columns, and bits. In *2016 IEEE International Test Conference (ITC)*, pages 1–7, November 2016. doi: 10.1109/TEST.2016.7805832.

[26] Rohan Hubli. 2.5d 3d ic market challenges and opportunities.

[27] H. Iwai. End of the scaling theory and Moore's law. In *2016 16th International Workshop on Junction Technology (IWJT)*, pages 1–4, May 2016. doi: 10.1109/IWJT.2016.7486661.

[28] Y. Jiang, X. He, C. Liu, and Y. Guo. An effective analytical 3d placer in monolithic 3d IC designs. In *2015 IEEE 11th International Conference on ASIC (ASICON)*, pages 1–4, November 2015. doi: 10.1109/ASICON.2015.7517146.

[29] Mitsumasa Koyanagi. 3d-IC Technology Using Ultra-Thin Chips. In Joachim Burghartz, editor, *Ultra-thin Chip Technology and Applications*, pages 109–123. Springer New York, New York, NY, 2011. ISBN 978-1-4419-7275-0 978-1-4419-7276-7. doi: 10.1007/978-1-4419-7276-7_11. URL `http://link.springer.com/10.1007/978-1-4419-7276-7_11`.

[30] J Lu, A Jindal, Y Kwon, J.J McMahon, M Rasco, R Augur, T.S Cale, and R.J Gutmann. Study of TSV Thinning Wafer Strength Enhancement for 3dic Package. *2003 IEEE International Interconnect Technology Conference (IITC)*, (03):74–76, 2003.

[31] E. J. Marinissen, B. De Wachter, K. Smith, J. Kiesewetter, M. Taouil, and S. Hamdioui. Direct probing on large-array fine-pitch micro-bumps of a wide-I/O logic-memory interface. In *2014 International Test Conference*, pages 1–10, October 2014. doi: 10.1109/TEST.2014.7035314.

[32] Erik Marinissen and Y Zorian. Testing 3d chips containing through-silicon vias. In *Proceedings - International Test Conference*, pages 1–11, December 2009. doi: 10.1109/TEST.2009.5355573.

[33] Erik Jan Marinissen. Creating options for 3d-SIC testing. In *VLSI Design, Automation, and Test (VLSI-DAT), 2013 International Symposium on*, pages 1–7. IEEE, 2013. URL `http://ieeexplore.ieee.org/abstract/document/6533800/`.

[34] Erik Jan Marinissen, Teresa McLaurin, and Hailong Jiao. IEEE Std P1838: DfT standard-under-development for 2.5d-, 3d-, and 5.5d-SICs. pages 1–10. IEEE, May 2016. ISBN 978-1-4673-9659-2. doi: 10.1109/ETS.2016.7519330. URL `http://ieeexplore.ieee.org/document/7519330/`.

[35] Gordon Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86, January 1998. URL `http://books.google.com/books?hl=en&lr=&id=I7o8teBhz5wC&oi=fnd&pg=PA56&dq=%22portable+communications+equipment.+The%22+%22will+be+more+powerful,+and+will+be%22+%22than+those+which+use+%E2%80%9Cconventional%E2%80%9D%22+%22was+to+miniaturize+electronics+equipment+to%22+%22high+reliability.+Even+at+the+present+level+of%22+&ots=Q5tor4F_EQ&sig=S5qwEBG8SfQrFl0dVB-9qy8JuYI`.

[36] Brandon Noia and Krishnendu Chakrabarty. *Design-for-Test and Test Optimization Techniques for TSV-based 3D Stacked ICs.* Springer Science & Business Media, November 2013. ISBN 978-3-319-02378-6. Google-Books-ID: biu_BAAAQBAJ.

[37] Zvi Or-Bach. Qualcomm to leverage monolithic 3d for smartphones. URL `http://www.eetimes.com/author.asp?doc_id=1326383`.

[38] Vinod Pangracious, Zied Marrakchi, and Habib Mehrez. Three-Dimensional Integration: A More Than Moore Technology. In *Three-Dimensional Design Methodologies for Tree-based FPGA Architecture*, volume 350, pages 13–41. Springer International Publishing, Cham, 2015. ISBN 978-3-319-19173-7 978-3-319-19174-4. doi: 10.1007/978-3-319-19174-4_2. URL `http://link.springer.com/10.1007/978-3-319-19174-4_2`.

[39] D. Patterson. The trouble with multi-core. *IEEE Spectrum*, 47(7):28–32, 53, July 2010. ISSN 0018-9235. doi: 10.1109/MSPEC.2010.5491011.

[40] R. S. Patti. Three-Dimensional Integrated Circuits and the Future of System-on-Chip Designs. *Proceedings of the IEEE*, 94(6):1214–1224, June 2006. ISSN 0018-9219. doi: 10.1109/JPROC.2006.873612.

[41] CIO Whitepapers Review. What is Cost Model - Definition and Explained. URL `https://whatis.ciowhitepapersreview.com/definition/cost-model/`.

[42] Breeta Sengupta, Urban Ingelsson, and Erik Larsson. Test Cost Modeling for 3d Stacked Chips with Through-Silicon Vias. page 4, 2011. URL `https://lup.lub.lu.se/search/publication/4302358`.

[43] G. Smith, L. Smith, S. Hosali, and S. Arkalgud. Yield considerations in the choice of 3d technology. In *2007 International Symposium on Semiconductor Manufacturing*, pages 1–3, October 2007. doi: 10.1109/ISSM.2007.4446880.

[44] Dylan Stow, Itir Akgun, Russell Barnes, Peng Gu, and Yuan Xie. Cost and Thermal Analysis of High-Performance 2.5d and 3d Integrated Circuit Design Space. pages 637–642. IEEE, July 2016. ISBN 978-1-4673-9039-2. doi: 10.1109/ISVLSI.2016.133. URL `http://ieeexplore.ieee.org/document/7560272/`.

[45] Chuan Tan, Ronald J. Gutmann, and L. Rafael Reif, editors. *Wafer Level 3-D ICs Process Technology.* Integrated Circuits and Systems. Springer US, Boston, MA, 2008. ISBN 978-0-387-76532-7 978-0-387-76534-1. doi: 10.1007/978-0-387-76534-1. URL `http://link.springer.com/10.1007/978-0-387-76534-1`.

[46] M. Taouil, S. Hamdioui, K. Beenakker, and E. J. Marinissen. Test Cost Analysis for 3d Die-to-Wafer Stacking. pages 435–441, December 2010. doi: 10.1109/ATS.2010.80.

[47] M. Taouil, S. Hamdioui, E. J. Marinissen, and S. Bhawmik. Impact of mid-bond testing in 3d stacked ICs. In *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, pages 178–183, October 2013. doi: 10.1109/DFT.2013.6653603.

[48] M. Taouil, S. Hamdioui, E. J. Marinissen, and S. Bhawmik. Using 3d-COSTAR for 2.5d test cost optimization. In *2013 IEEE International 3D Systems Integration Conference (3DIC)*, pages 1–8, October 2013. doi: 10.1109/3DIC.2013.6702351.

[49] M. Taouil, S. Hamdioui, and E. J. Marinissen. Quality versus cost analysis for 3d Stacked ICs. In *2014 IEEE 32nd VLSI Test Symposium (VTS)*, pages 1–6, April 2014. doi: 10.1109/VTS.2014.6818763.

[50] Mottaqiallah Taouil. *Yield and cost analysis or 3D stacked ICs.* PhD thesis, [s.n.], S.l., 2014. OCLC: 905871299.

[51] Mottaqiallah Taouil and Erik Jan Marinissen. 3d-COSTAR: Analytical Cost Formulation of 2.5d-SICs and 3d-SICs. page 15.

[52] Mottaqiallah Taouil, Said Hamdioui, and Erik Jan Marinissen. On modeling and optimizing cost in 3d Stacked-ICs. In *Design and Test Workshop (IDT), 2011 IEEE 6th International*, pages 24–29. IEEE, 2011. URL `http://ieeexplore.ieee.org/abstract/document/6123096/`.

[53] Mottaqiallah Taouil, Mahmoud Masadeh, Said Hamdioui, and Erik Jan Marinissen. Interconnect test for 3d stacked memory-on-logic. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 126. European Design and Automation Association, 2014. URL `http://dl.acm.org/citation.cfm?id=2616761`.

[54] Tezzaron. Our technology for non-technical folks. URL `https://tezzaron.com/ourvision/our-technology-for-non-technical-folks/`.

[55] Neil Thompson. Moore's Law Goes Multicore: The economic and strategic consequences of a fundamental change in how computers work. page 2.

[56] Waferworld. Silicon wafer | what you need to know about silicon wafers. URL `https://www.waferworld.com/silicon-wafer-about/`.

[57] Y. Yang, M. Yu, Rusli, Q. Fang, J. Song, L. Ding, and G. Q. Lo. Through-Si-via (TSV) Keep-Out-Zone (KOZ) in SOI Photonics Interposer: A Study of the Impact of TSV-Induced Stress on Si Ring Resonators. *IEEE Photonics Journal*, 5(6):2700611–2700611, December 2013. ISSN 1943-0655. doi: 10.1109/JPHOT.2013.2285707.

[58] L. Zhang, H. Y. Li, G. Q. Lo, and C. S. Tan. Thermal characterization of TSV array as heat removal element in 3d IC stacking. In *2012 IEEE 14th Electronics Packaging Technology Conference (EPTC)*, pages 153–156, December 2012. doi: 10.1109/EPTC.2012.6507069.