UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MARCELO CORRÊA DE BITTENCOURT

Comparing Different And-Inverter Graph Data Structures

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. André Inácio Reis

Porto Alegre
November 2018

CIP — CATALOGING-IN-PUBLICATION

de Bittencourt, Marcelo Corrêa

Comparing Different And-Inverter Graph Data Structures / Marcelo Corrêa de Bittencourt. – Porto Alegre: PPGC da UFRGS, 2018.

66 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2018. Advisor: André Inácio Reis.

1. AIG. 2. Testing. 3. Performance testing. 4. Data Structures. I. Reis, André Inácio. II. Título.

"When you talk, you are only repeating what you already know.

But if you listen, you may learn something new."

— Dalai Lama XIV

ACKNOWLEDGMENTS

Gostaria de agradecer primeiramente aos meus pais, que desde muito cedo foram incansáveis na tarefa de garantir meu acesso a toda a educação formal que estivesse ao alcance da família. Sem esse apoio desde os anos iniciais de estudo eu não teria chegado até aqui.

Agradeço a minha esposa Paula por todo o suporte e compreensão ao longo desta jornada do mestrado, principalmente nos momentos em que sacrifiquei momentos de convivência em detrimento do foco no mestrado. Sua ajuda, apoio e conselhos tornaram a caminhada mais previsível e clara.

Quero agradecer também aos meus colegas do Logics por todas as dicas, trocas de idéias, críticas e sugestões, especialmente ao colega Vinicius Possani que compartilhou informações relevantes sobre testes de desempenho. Isso foi muito importante para que eu construisse o caminho até aqui com sucesso.

Agradeço ao colega de área Tiago Fontana, da UFSC, por ter explicado detalhadamente como utilizou e aplicou testes de desempenho em seus experimentos, especialmente com a ferramenta PAPI. Seu auxílio foi vital para que eu tivesse sucesso ao rodar os experimentos coletando dados de um trecho de código específico.

Também agradeço os meus sogros pelo apoio, palavras positivas e compreensão pela minha ausência para que eu pudesse me dedicar a esta fase de estudos.

Gostaria de agradecer também aos meus amigos e colegas de trabalho por toda a força positiva que me transmitiram. Isso ajudou a fortalecer minha persistência em concluir esse trabalho que resume a minha dedicação ao mestrado.

ABSTRACT

This document presents a performance analysis of four different And-Inverter Graph (AIG) implementations. AIG is a data structure commonly used in programs used for digital circuits design. Different implementations of the same data structure can affect performance. This is demonstrated by previous works that evaluate performance for different Binary Decision Diagram (BDD) packages, another data structure widely used in logic synthesis. We have implemented four distinct AIG data structures using a choice of unidirectional or bidirectional graphs in which the references to nodes are made using pointers or indexed using non-negative integers. Using these different AIG data structures, we measure how different implementation aspects affect performance in running basic algorithm.

Keywords: AIG. Testing. Performance testing. Data Structures.

Comparativo de diferentes estruturas de dados de And-Inverter Graph

RESUMO

Este documento apresenta uma análise de desempenho de quatro diferentes implementações de And-Inverter Graph (AIG). AIGs são estruturas de dados normalmente utilizadas em programas que são utilizados para design de circuitos digitais. Diferentes implementações da mesma estrutura de dados pode afetar o desempenho. Isto é demonstrado em trabalhos anteriores que avaliam o desempenho de diferentes pacotes BDD (Binary Decision Diagram), que é outra estrutura de dados largamente utilizada em síntese lógica. Foram implementadas quatro estruturas de dados diferentes utilizando grafos unidirecionais ou bidirecionais aos quais os nodos são referenciados utilizando ponteiros ou índices de inteiros não-negativos. Utilizando estas diferentes estruturas de dados de AIG, medimos como diferentes aspectos das implementações afetam o desempenho da execução de um algoritmo básico.

Palavras-chave: AIG, Teste, Teste de desempenho, Estruturas de dados.

LIST OF ABBREVIATIONS AND ACRONYMS

EDA	Electronic Design Automation
AIG	And-Inverter Graph
BDD	Binary Decision Diagram
BFS	Breadth First Search
CM	Cache Misses
DFS	Depth First Search
IB	Index-based bidirectional data structure
IU	Index-based unidirectional data structure
LBBDD	Lower Bound Binary Decision Diagram
MTM	More Than Million benchmark
PB	Pointer-based bidirectional data structure
PU	Pointer-based unidirectional data structure
ROBDD	Reduced Ordered Binary Decision Diagram
TSBDD	Terminal Supressed Binary Decision Diagram
XAIG	Xor-And-Inverter Graph

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

This work focuses on the field of EDA, or Electronic Design Automation. More specifically, we study AND-Inverter Graphs, a data structure that is widely used in the logic synthesis of digital VLSI circuits, such as arithmetic circuits (OSORIO et al., 2004) used in microprocessors where computer operating systems are built (ROSA et al., 2003).

1.1 Context and works from the Logics group

The Logics group at UFRGS has a long tradition of working in the field of EDA. Early works include approaches for library free technology mapping (REIS; ROBERT; REIS, 1998) (REIS, 1999) (CORREIA; REIS, 2004) (MARQUES et al., 2007), where a circuit is mapped directly to transistor networks (SCHNEIDER et al., 2005) (POSSANI et al., 2012) (POSSANI et al., 2013) (POSSANI et al., 2016) (MARTINS et al., 2011) without using pre-defined (and pre-validated) cell libraries (REIS; ANDERSON, 2011) (REIS et al., 2011) (BEM et al., 2011) (RIBAS et al., 2011) (REIS, 2012) (MARTINS et al., 2015). Mapping directly at the transistor level can produce circuits that are more efficient (SCHNEIDER; RIBAS; REIS, 2006). In order to produce efficient circuits at the transistor level, it is necessary to produce good transistor networks with respect to power consumption (BUTZEN et al., 2007) (BUTZEN et al., 2010) (WILTGEN et al., 2013) and reliability (SILVA; REIS; RIBAS, 2009) (SILVA; REIS; RIBAS, 2010) (BUTZEN et al., 2010) (BUTZEN et al., 2012), including networks for memory cells (NUNES et al., 2013). Digital circuits can be synchronous (MARQUES et al., 2005) or asynchronous (MOCHO et al., 2006) (MOREIRA et al., 2014), and this work will focus on synchronous circuits.

The Logics group also introduced the functional composition method for logic synthesis (MARTINS et al., 2010), which is used in applications for novel technologies such as threshold logic (NEUTZLING et al., 2013), memresistive material implication stateful logic (MARRANGHELLO et al., 2015), and redundant circuits with TMR (GOMES et al., 2014) (GOMES et al., 2015).

Logic synthesis has a wide range of applications, from the basic methods to simplify equations (KLOCK et al., 2007) to more efficient methods with the same goal (MARTINS et al., 2010). In this work I will focus on evaluating different data structures for AIGs for large circuits, as proposed by my advisor, in the context of the work developed at the UFRGS Logics Group.

1.2 Context and And-Inverter Graphs

In integrated circuit technology latest advances implies the need to treat circuits with several million logic gates. At the logic level, And-Inverter Graphs (AIGs) are one of the data structure used to manipulate digital circuits. For this reason, scalability in memory usage and execution time is of paramount importance. In order to have data structures that provide support for high performance algorithms, it is necessary to understand (1) what are the possible variations for the data structure; and (2) how to test the software implementation of the data structure, including memory usage, performance and other tests.

Concerning possible variations in the data structure, it is known that different implementations of the same data structure can affect performance. This is demonstrated by previous works (SENTOVICH, 1996; MANNE; GRUNWALD; SOMENZI, 1997; YANG et al., 1998; JANSSEN, 2003; JANSSEN, 2001) that evaluate performance for different Binary Decision Diagram (BDD) packages, another data structure widely used in logic synthesis. BDDs have been tested regarding performance issues in several studies (SENTOVICH, 1996; MANNE; GRUNWALD; SOMENZI, 1997; YANG et al., 1998; JANSSEN, 2003).

The work of Sentovich (SENTOVICH, 1996) made the first study of performance for BDD packages. Manne et al. (MANNE; GRUNWALD; SOMENZI, 1997), started to investigate how memory locality influenced performance through cache hits and misses. The work of Yang et al. (YANG et al., 1998) investigated the performance of BDD packages when applied to formal verification of digital circuits. A pointerless BDD package has been proposed by Janssen (JANSSEN, 2001), aiming to exceed the performance of a pointer based package using a index-based reference for nodes. In a subsequent work, Janssen (JANSSEN, 2003) compared several BDD packages, including his pointerless package. Despite all these studies, similar studies are lacking for AIG implementations.

Concerning software testing for different data structures, testing issues are not well explored in conjunction with the EDA research field. The work of Wallace and Bloom (WALLACE; BLOOM, 2012) advocated the use of several types of software test to produce more reliable code, specially the use of the so-called unit testing to deal with legacy code that is so common in the EDA industry. In the logic synthesis field, specially related to EDA tools, the testing research is focused in robustness and reliability from a user point of view (PUGGELLI et al., 2011; SCHMIDT; FIER; BALCáREK, 2014). The type of tests presented investigate how different equivalent input descriptions affect performance and reliability. The use of different inputs for software tests is studied by Ammann and Offutt (AMMANN; OFFUTT, 2017), where inputs are cataloged according to some characteristics. This approach is called input space partitioning [1], in the sense that the input space is partitioned according to the characteristics, in order to include inputs with varied characteristics (i.e. belonging to different partitions of the input space). Given that BDD has a substantial number of research papers regarding performance, in this dissertation we will contribute with the performance analysis of other common used data structure: AIG.

In this dissertation we will analyze and show the performance results of four different implementations of AIGs written specifically for this dissertation: pointer-based unidirectional, pointer-based bidirectional, index-based unidirectional and index-based bidirectional data structure. The pointer and index-based implementations were chosen to investigate the impact of data locality, focusing in runtime and cache misses tests. The unidirectional and bidirectional implementations were chosen to investigate the impact of memory usage given that bidirectional implementations have additional directional information. There are other possible different implementations of AIGs, for example, adjacency list, adjacency matrix or incidence list that are out of scope of this work given that the four chosen implementation are enough to have an initial investigation of data locality and memory usage impact. These out of scope implementations could be used to expand the investigation of different criteria in a further work.

The criteria of runtime and memory usage was chosen as a performance measure given that papers that analyze BDD performance (SENTOVICH, 1996; JANSSEN, 2001) already used these two criteria to investigation. Papers focused in investigation of data locality (MANNE; GRUNWALD; SOMENZI, 1997; YANG et al., 1998) use cache misses criteria beyond the two previous performance criteria. Some works (FONTANA et al., 2017a; FONTANA et al., 2017b) use just runtime and cache misses criteria in the results. Considering the use of criteria runtime, memory usage and cache misses in these previous related papers to investigate performance and data locality, we decided to use these same criteria because they suit our needs of investigation.

Another issue that can influence the results of EDA algorithms are external bias,

[1]The book's authors present two input domain modeling approaches: interface-based and functionality-based

for example, studied by Fiser (FIšER; SCHMIDT; BALCáREK, 2014) where it was showed that permuted inputs can influence the results of EDA algorithms. In our experiments we validated this external bias using different versions of AIG files with AND nodes with the tag name defined following different orders: BFS and DFS. Given that we did not find any influence in the execution of the path algorithm changing the tag name in both different order, we decided to use only a fixed order of definition of tags for node and focus the experiments to test runtime, memory usage and cache misses.

1.3 Organization of this work

Regarding the general work organization, in the chapter 2 we describe BDDs and the evolution of papers regarding BDD performance issues concerned to pointer-based and pointerless implementations, use of BDD in related works and then situating the goal of this work regarding AIGs performance. In the next chapter we describe AIGs, the AIGER format for AIG specification, use of AIG in related works and specific issues related to AIGs necessary to understand the different implementations including the synthetic AIGs. The four data structure implementations are described and their differences are compared in the chapter 4. In the chapter 5 we present the information about the computer used in the experiments, memory hierarchy, memory management, and performance tools used in this work to collect the results. The experiment description, the basic algorithm, and the results are presented in the chapter 6, showing the performance of the four different implementations regarding runtime, memory usage and cache misses. The chapter 7 is dedicated to the final analysis of the work pointing out the limitations and future work.

2 STATE OF THE ART

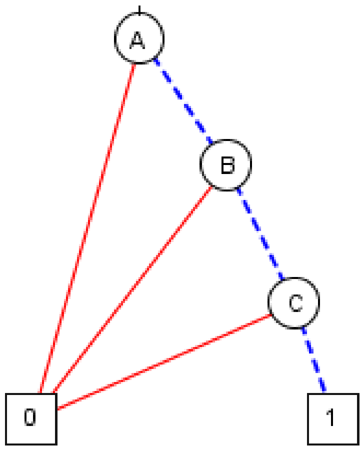
This chapter presents the evolution of studies of representation of circuit using BDD from pointer implementation until the pointerless implementation. We list some relevant work that use BDD. In the sequence, we point out the advantages of the pointerless BDD implementation. We conclude this chapter positioning the contribution of this work in the scenario of investigation of performance results of another popular data structure circuit representation: AIG.

2.1 BDD

Binary Decision Diagram is the type of graph or diagram that can be used to represent a binary function. Variables are represented by nodes and the variable values are represented by edges. It is read from the top to the bottom. Two edges are derived from each node: one for the 0 binary value and the other for the 1 binary value path. In the base of the BDD we have one square representing the 0 value and another square for the 1 value. A complete path from the top to the base represents a combination of values that will end in the result found in the respective square (0 or 1) in the graph base (WAGNER; REIS; RIBAS, 2006).

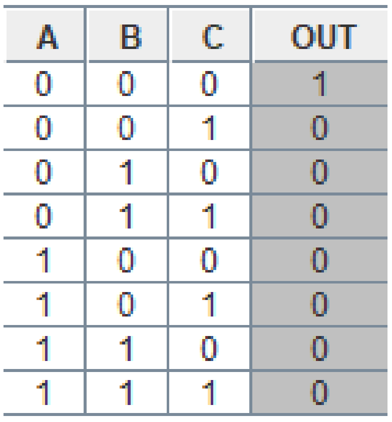
BDDs are considered an efficient data structure for some applications, specially many EDA tools in the area of formal verification (JANSSEN, 2001). Regarding the data structure to represent the BDD considering the Fig. 2.1 each arc is a pointer in the data structure and each node has two pointers. The truth table for BDD in Fig. 2.1 can be seen in Fig. 2.2. Given that we have three nodes in this BDD in Fig. 2.1, using the pointer-based representation, we would need six times the size occupied to store a memory address in the architecture to represent this BDD in memory.

Figure 2.1: An example of a valid BDD.



Source: André Reis

Figure 2.2: BDD And2 Truth table of the test Fig. 2.1.



A	B	C	OUT
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Source: André Reis

Concern about BDD performance leaded to previous works (SENTOVICH, 1996; MANNE; GRUNWALD; SOMENZI, 1997) that demonstrate that differences in data structure implementation can affect BDD performance. Sentovich et al. (SENTOVICH, 1996) qualitatively compares ROBDDs that have complement edged and an efficient memory management of intermediate results and unified storage for each variable. These are considered state-of-art ROBDDs by the author. Time and memory usage are showed in the results of this study. Manne et al. (MANNE; GRUNWALD; SOMENZI, 1997) conducted an analysis of BDD packages using BFS and DFS approaches quantitatively measuring the BDD packages CMU, CAL and CUDD. The results are showed in terms of execution time, memory usage and percentage of cache miss rates. They showed the positive effects of improvements in memory locality in DFS packages.

2.2 Use of BDD related works

Reis et al. propose a class of BDDs that performs library free technology mapping overcoming previous difficulties and allowing direct association of transistors with BDD arcs (REIS et al., 1995b). In this sense, when the number of arcs is minimized in this new class called TSBDD, the same minimization in number of transistors can be implemented when dealing with physical implementation.

Reis et al. address the design of integrated circuits without using cell libraries, using instead CMOS complex gates as a mean to get better performance results. The library free method proposed is based on the novel class of BDD, the so-called TSBDD (REIS et al., 1995a). This class enables direct association of arcs and transistors, controlling the overall number of transistors, and the number of serial transistors. The results showed that a good trade-off can be get in terms of power, area and delay when using CMOS complex gates instead of simple gates. The generation of logic circuits not based in cell libraries eliminates the necessity of technology mapping.

Reis et al. (1997) use this new class TSBDD as a base for a method for mapping a set of boolean equations into a set of Static CMOS Complex Gates (SCCGs) (REIS et al., 1997). The tool TABA that implements this method was built and the experimental results in this work demonstrated that this applied method delivers more reduction in overall number of transistors compared to the published results in previous works for Library Free Technology Mapping.

Correia and Reis (2001) choose BDDs as the data structure to be used in the verification of function equivalence based on the criteria that when the work was written BDD was the most used form to represent boolean function in EDA (CORREIA; REIS, 2001). In this work, the boolean functions are classified in P classes, where the inputs can be permuted in place. The result is the same and NPN classes using inverters in inputs or outputs will have the same result of other functions. It concludes that the criteria of number of P classes is the best to use because P permutation is always allowed. The NPN classes will depend on the cost of use of negation. A tool based on the special type of BDD, called ROBDD was written for this work. This type of BDD implements strong canonical boolean functions. The tool groups functions into P and NPN classes was built. The goal of this tool is to enumerate the number of P and NPN classes of n-input functions.

Togni et al. present a tool that implements the functionality that can generate cell libraries from boolean functions or truth tables (TOGNI et al., 2002). This higher level of abstraction to generate cell layout is not available in other tools, that uses as input Symbolic Layout (Cadabra) or Spice description (Prolific). Boundary can be informed to generate the cell layout: maximum number of cell inputs and maximum number of serial transistors. It leverage a combination of logical and physical synthesis. BDDs are used as inputs to generation of pass transistor logic in the transistor level cell description.

Poli et al. propose an unified method to build cell-level transistor networks from BDDs (POLI et al., 2003). Another method to build these networks consists in mapping a circuit into logic cells creating a library of cells. So on, these cells, that have their cost metric defined, are placed and routed forming the final circuit. The unified method proposed presents better results using transistor count as the key cost metric when comparing with other methods that generate a complete library composed of all four-input functions.

Junior et al. (2006) compare different ways to derive transistor networks from BDDs (JUNIOR et al., 2006). The experiment results of using disjoint pull-up and pull-down planes demonstrates shorter pull-up and pull-down transistor stacks. It is used unateness reduction that leads to faster circuits thanks to generation of near PTL logic.

Junior et. al (2007) compare two different CMOS logic styles, one derived from sum-of-products equations and another from ROBDD graphs (JUNIOR et al., 2007). The results are expressed in transistor count and logical effort. Related to BDD, in this work some methods use BDDs as a base to generate transistor networks. Some methods are based in multiplexer-based logic. Another uses direct switch association to BDD arcs to build Logic Cells BDD-based networks, called LBBDDs.

Junior et al. (2008) propose a new static and disjoint logic style called LBBDD (JUNIOR, 2008). Comparing to NCSP, it has the advantage that no factorization is necessary to achieve optimized networks given that optimization are performed previously in the BDD structure. This logic minimizes the total transistor count.

Rosa et. al (2009) compared six different approaches that generate transistor networks (ROSA et al., 2009). The method that generates transistor networks from BDDs duplicates some nodes of the BDD to avoid the insertion of sneak paths in the BDD and forced the achievement of the lower bounds. The application of this method results in the reduction of the longest transistor chain, conducting to the achievement of the lower bound.

Reis et. al (2010) propose a method to optimize a design and a library choosing the best set of cells to implement the design (REIS et al., 2010). The maximum number of arcs in series in a BDD implementation of the function can be used to implicitly define the additional set of admissible functionalities. BDD height representing the functionality can be used to accept or discard functionalities when defining a library.

Martins et al. (2013) use the root node of a binary decision diagram (BDD) or an integer to represent the function in the tuple function, spin diode logic tree from the bonded-pair representation (MARTINS et al., 2013). This representation is an adaptation of the boolean factoring algorithm using functional composition (fc) to synthesize a spin diode network. Callegaro et al. propose an unatization algorithm that uses the data structure reduced and ordered binary decision diagram (ROBDD) as its base (CALLEGARO et al., 2013).

2.3 Pointer-based and pointerless BDD packages

When machines passed from 16 bits to 32 bits two advantages could be pointed out: more memory were able to be addressed and then larger BDDs could be treated. However pointers passed from 16 bits to 32 bits and then BDDs had doubled their size with 32 bits. In the work of Yang et al. (YANG et al., 1998) it was pointed out this concern in the machines of the generation after 32 bits, with memory size greater than 4GB, that will double again the pointer size, in this case, from 32 bits to 64 bits.

Some BDD models were checked by Yang et al. (YANG et al., 1998). The authors studied the effects of the BFS approach in memory locality, considering the cache hit rate. They saw as a positive indication that index-based approach could lead to a solution to this memory overhead problem, given that ABCD BDD package was the best in the study and it uses index-based reference.

Aiming to solve some limitation of pointer-based implementations of BDD, a pointerless BDD package was proposed by Janssen (JANSSEN, 2001). This new proposed approach has same advantages comparing to pointer-based approach: size of integers can be controlled easily, by changing the type of integers; integers are allocated by the program, not by the operating system; behavior does not change with operating system; integers (32 bits) have smaller size than pointers (64 bits).

In logic synthesis the most used data structure is currently the And-Inverter Graphs. Giving that AIG lacks similar studies of performance of pointer and pointerless implementations, our work is to measure how efficient is the index-based implementations compared to pointer-based implementations using the AIG data structure.

3 AND-INVERTER GRAPHS

In this chapter we present And-Inverter Graphs (AIGs), the graphs that we investigate in this dissertation, concerning different data structures for representation. This chapter provides the necessary background to understand this type of graphs.

3.1 AIGs and AIGER format

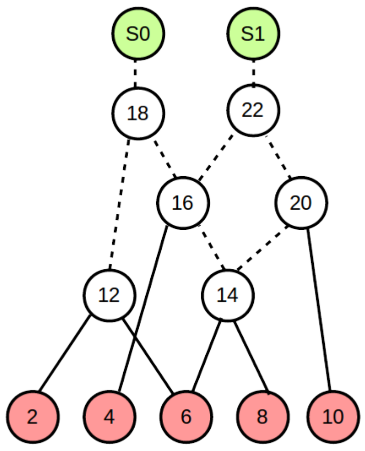
And-inverted graphs (AIGs) are a type of direct acyclic graphs (DAGs) widely used in logic synthesis. AIGs are composed of three types of nodes: input nodes, output nodes or two-input AND nodes. A given node can have 0 or 2 incoming edges. Nodes without incoming edges are considered primary inputs. Nodes with 2 incoming edges are two-input AND nodes. Some nodes are marked as outputs. Edges can be direct or complemented to indicate when a signal is inverted (MISHCHENKO; CHATTERJEE; BRAYTON, 2006). Nodes marked as outputs do not have out-coming nodes.

A well established AIG format is defined by Biere (BIERE, 2007). This format has been successfully used in the popular academic tool ABC (Berkeley Logic Synthesis and Verification Group, 2018; BRAYTON; MISHCHENKO, 2010) and in the Simple Flow Synthesis tool (LogiCS Research Lab, 2017; MATOS, 2017). The AIGER AIG format specifies an ASCII format and a binary format. This format is supported by an AIG library with tools to work with AIGER files. This library provides utilities such as a translator from ASCII to binary format and vice versa. An example of AIG for the C17 benchmark circuit is shown in listing 3.1. The corresponding AIG graph is shown in Fig. 3.1.

Listing 3.1 – Text description of the C17 benchmark in AIG format.

```
aag 11 5 0 2 6
2
4
6
8
10
19
23
12 6 2
14 8 6
16 15 4
18 17 13
20 15 10
22 21 17
i0 A
i1 B
i2 C
i3 D
i4 E
o0 S0
o1 S1
c
C17
```

Figure 3.1: C17 benchmark viewed as an AIG Graph.



Source: André Reis

The first line in listing 3.1 is a line containing information about the number of elements in the AIG. This first line is called header. An AIGER file in ASCII format starts with the format identifier string ’aag’ for ASCII AIG and 5 non negative integers ’M’, ’I’, ’L’, ’O’, ’A’ separated by spaces. The interpretation of the ’MILOA’ integers is as follows. The maximum variable index is given by M=11 in the header in listing 3.1. As in AIGs, variables are coded as integers, the maximum variable index can be used to compute the maximum integer needed to represent AIG signals. The maximum integer can be computed as Mi=2*M+1; Mi=23 in our example considering that the node with the higher tag number, tag 22, can be negated (inverted), and then 22 (non inverted) would be represented by 23 (inverted). This way, if vectors are used to represent the AIG, the vector has to allocate until the maximum position Mi=23.

The number of inputs is given by I=5 in the header of listing 3.1. Meaning that our example AIG has five distinct inputs. This can be observed in listing 3.2, where the inputs 2, 4, 6, 8, 10 are listed one by line. Afterwards they are attributed the labels A, B, C, D, E, one by line.

Listing 3.2 – Text description corresponding to the inputs of the C17 benchmark in AAG format.

```
...
2
4
6
8
10
...
i0  A
i1  B
i2  C
i3  D
i4  E
...
```

The number of latches is given by L=0 in header of listing 3.1. Latches will not be used in this dissertation. We will restrict our experiments to combinational circuits without latches.

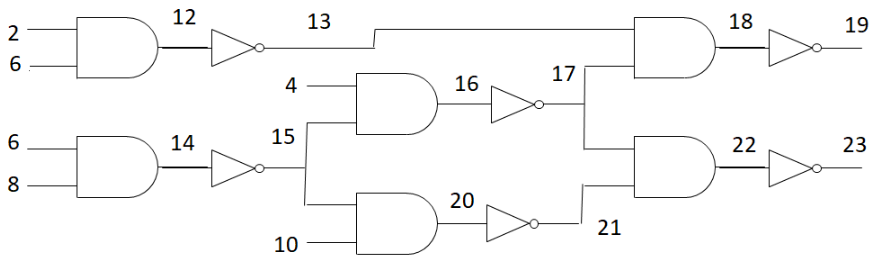
The number of outputs is given by O=2 in header of listing 3.1. Meaning that our example AIG has two distinct outputs. This can be observed in listing 3.3, where the inputs 19, 23 are listed one by line. Afterwards they are attributed the labels S0, S1, one by line.

Listing 3.3 – Text description corresponding to the Outputs of the C17 benchmark in AAG format.

```
...
19
23
...
o0  S0
o1  S1
...
```

The number of AND gates is given by A=6 in listing 3.4. This means that six nodes in the AIG in listing 3.1 are AND gates. These are the nodes labeled 12, 14, 16, 18, 20, 22 in listing 3.4. The line 12 6 2 represents an AND gate whose output is variable

Figure 3.2: C17 benchmark viewed as a digital circuit.



Source: André Reis

12 and inputs are nodes 6 and 2. The line 18 17 13 represents an AND gate whose output is variable 18 and inputs are nodes 16 (but inverted, so referenced as 17) and 12 (but also inverted, so represented as 13).

Listing 3.4 – Text description corresponding to the Ands of the C17 benchmark in AAG format.

```
...
12  6  2
14  8  6
16  15  4
18  17  13
20  15  10
22  21  17
...
```

Fig. 3.2 presents a circuit version of C17 benchmark. Notice that it has five inputs (I=5), six AND gates (A=6) and two outputs (O=2). Also, all the odd numbers are output of inverters, as odd numbers represent inverted signals. In the AIG graph this is represented as dotted lines. In a circuit, inverters are explicit and always the input is an even number and the output is an odd number, according to aiger convention for signals.

3.2 Use of AIG related works

Martinello et al. introduce the concept of kl-feasible cuts, that controls the number of inputs (k) and as a contribution, the number of outputs (l) (MARTINELLO et al., 2010). The concept of factor cuts is extended to kl-cuts, providing scalability. It is presented a novel algorithm that computes kl-feasible cuts with unbounded k using as data structure AIG. Figueiro, Ribas and Reis propose a technology independent synthesis algorithm that uses AIG data structure resulting in a smaller number of nodes and logical depth

(FIGUEIRÓ; RIBAS; REIS, 2011). This novel approach for AIG construction is based on a new synthesis paradigm called function composition.

Martins, Ribas and Reis present a new paradigm for combinational logic synthesis called functional composition (MARTINS; RIBAS; REIS, 2012). This method decomposes a boolean function into a set of smaller functions that implements it. A functional composition algorithm variant for AIG rewriting is presented. The BDDs in these work serves as a canonical representation of the function using ROBDDs for bounded pairs representing logic functions. Concerning AIG rewriting, the bonded-pair representation is a pair of <fr, sr>, where the functional representation (fr) is an integer array or a BDD node. AIG node and structural information on the rooted AIG are contained in the structural representation (sr). In AIG rewriting the functional composition minimizes a cost functions, usually the number of nodes or the graph height while constructing AIGs from simple graphs. This method presents improvements in the number of nodes when compared to different approaches ABC + FRAIG and FC factoring + FRAIG. Improvement in logical depth is get when comparing to FC factoring + FRAIG.

Machado et at. present an approach to enumerate kl-cuts on top of mapped circuits (MACHADO et al., 2012). Comparing to kl-cuts on top of AIGs it was possible to improve the circuits and replace them in the original circuits, aiming to improve delay and area of mapped circuits, where preliminary results showed improvement in 19% and 24% in area and delay respectively.

Neutzling et al. (2015) propose a novel approach to synthesize circuits based on threshold logic gates (TLGs) (NEUTZLING et al., 2015). the first task in this approach is to compute priority cuts in the input AIG. Using AIGs as subject graph gives the advantage that the entire design is decomposed into AND nodes, which are threshold logic functions.

Matos proposes a set of graph-based algorithms for efficiently mapping VLSI circuits using simple cells (MATOS, 2017). It is presented a sequence of phases until reach the last phase that will delivery a more efficient circuit. The phases are: improvements in AIG node count minimization, XAIG generation, deriving polarity graphs from XAIGs, improvements in the graph coloring procedure, deriving optimized circuits from colored polarity graphs and the are-oriented level-aware fanout limitation. As secondary contributions according to the author, four contributions are presented: two of them related to synthesis speed up called merging locally equivalent flip-flops and exploring two-output flip-flops, and another two related to improvements to the quality-of-results (QoR) called improving the QoR with trade-off optimizations and improving runtime with parallel synthesis respectively. The results showed improvements in average number of inverters, transistors, area, power dissipation and delay when comparing with academic and commercial approaches.

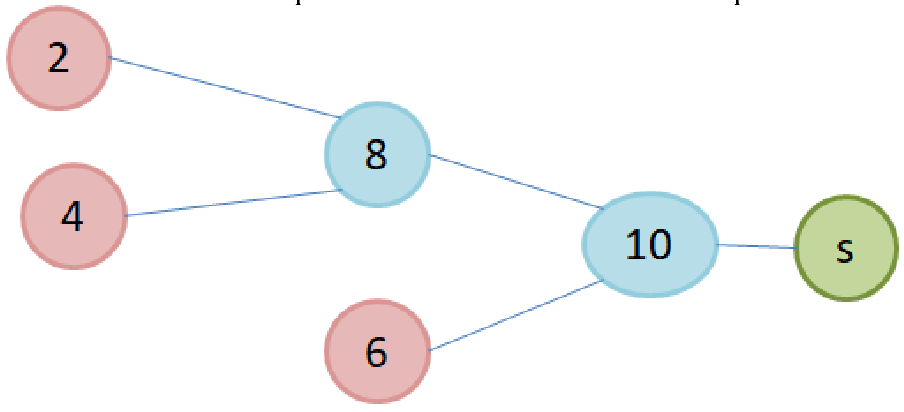
3.3 Labeling the nodes of AIGs

Labeling of AIG nodes must follow some strict rules, such that the number tags are attributed in a way that guarantees the number attributed to outputs of a node are always larger than the number attributed to inputs of the node. The next subsections will discuss valid and invalid orders; then we will discuss alternative different valid orders for numbering an AIG.

3.3.1 Valid and invalid orders for labeling the nodes of AIGs

Fig. 3.3 shows a valid order for numbering the nodes of an AIG. Notice that the C17 benchmark illustrated in listing 3.1 and in Fig. 3.1 also presented a valid numbering of nodes. AIGs with a valid order must have node numbering such that inputs have smaller number than outputs.

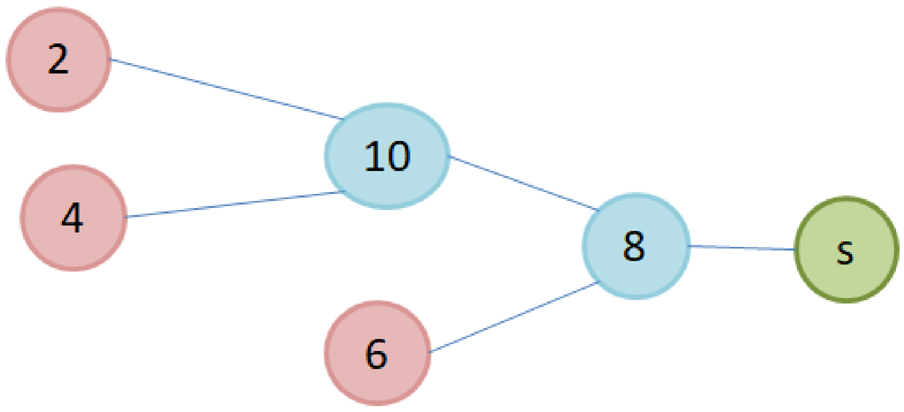
Figure 3.3: AIG such that inputs have smaller number than outputs: a valid order.



Source: André Reis

Fig. 3.4 shows an invalid order for numbering the nodes of an AIG. The order is invalid because the number of the output is smaller than the number of one of the inputs.

Figure 3.4: AIG such that one output have smaller number than one input: an invalid order.

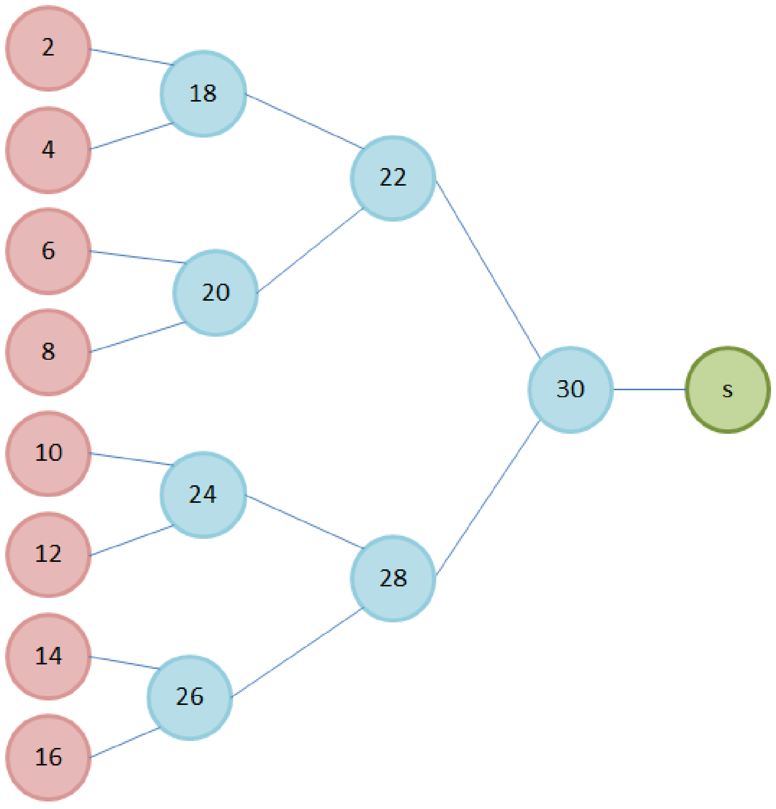


Source: André Reis

3.3.2 Alternative valid orders for labeling the nodes of AIGs

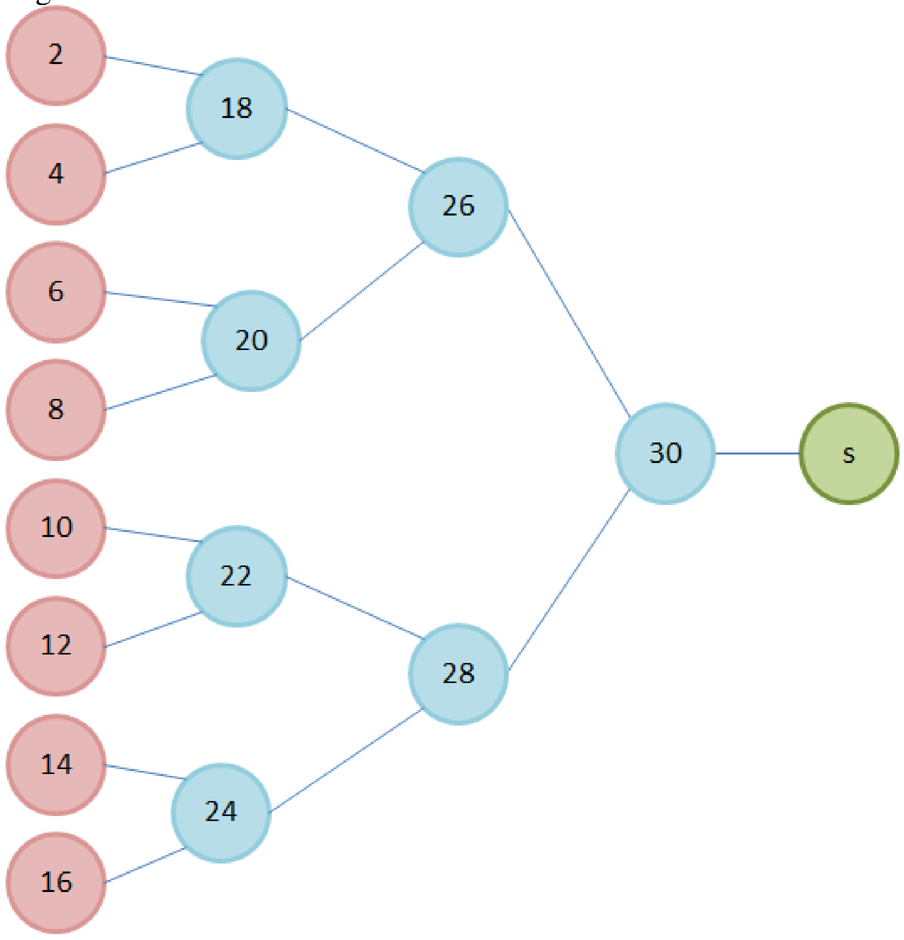
It is important to notice that there is more than one valid order for numbering the nodes of an AIG. One possibility is to number the variables of the nodes in Depth-First-Search (DFS) order, as shown in Fig. 3.5. Another possibility is to number the variables of the nodes in Breath-First-Search (BFS) order, as shown in Fig. 3.6.

Figure 3.5: A circuit where variables were numbered with DFS order.



Source: André Reis

Figure 3.6: A circuit where variables were numbered with BFS order.



Source: André Reis

3.4 Synthetic AIG benchmarks

In the scope of this dissertation we use synthetic AIG benchmarks. The benchmarks are perfect balanced binary trees from 10 to 25 levels of nodes. As binary trees, the number of nodes is calculated by the formula $n = (2^L) - 1$, where L is the number of levels. The number of inputs is calculated by the formula $i = 2^{(L-1)}$. The respective levels, nodes and number of inputs for each synthetic benchmark is presented in Table 3.1. These synthetic benchmarks have an important difference when compared to regular benchmarks that have millions of nodes: the number of inputs. Regular benchmarks normally have a maximum of 800 inputs. Synthetic benchmarks have as much inputs as the number of levels as we saw in the previous formula. So, it can be seen in Table 3.1 that since level 11, the synthetic benchmarks have more than 800 inputs. Synthetic benchmarks were used to exercise the data structures but they lack features found in real circuits. Regarding the limits of the synthetic benchmarks considering that each node has

Table 3.1: Number of levels and nodes for the synthetic benchmarks which are complete binary trees.

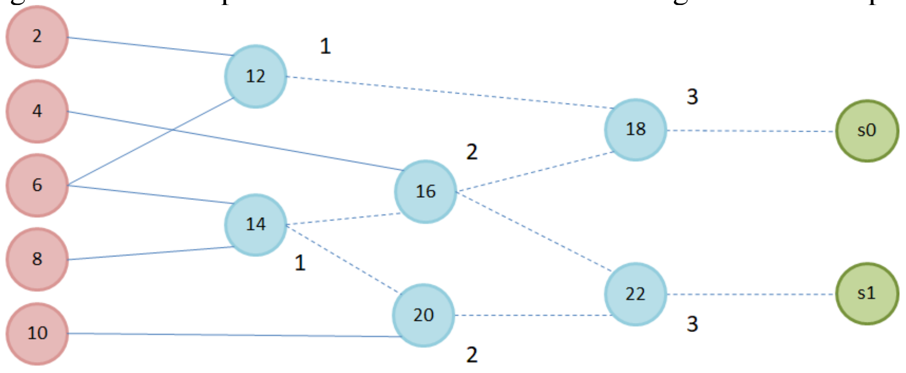
Levels	Nodes	Inputs
10	1,023	512
11	2,047	1,024
12	4,095	2,048
13	8,191	4,096
14	16,383	8,192
15	32,767	16,384
16	65,535	32,768
17	131,071	65,536
18	262,143	131,072
19	524,287	262,144
20	1,048,575	524,288
21	2,097,151	1,048,576
22	4,194,303	2,097,152
23	8,388,607	4,194,304
24	16,777,215	8,388,608
25	33,554,431	16,777,216

a different even number (starting with 2) as a tag when it is not inverted or it has the next odd number when it is inverted, the node tag was implemented as unsigned integer and 32 bits were reserved to represent integers, it will be available $(2^{32}) - 1$ integers, that corresponds to 4,294,967,295. Given that each node is represented by 2 integers (even or odd depending on the inversion), it is possible to allocate 2,147,483,647 nodes in these conditions.

3.5 Computing Depths on AIGs

A simple AIG task is to compute the worst case depth of each AIG node. Fig. 3.7 shows the worst case depths for an example AIG. The algorithm to compute these depths is trivial. In this dissertation we implement this algorithm over four distinct data structures for AIGs. Our goal is to verify how the data structures affect the performance of the algorithms and this way to assert the quality of the data structures.

Figure 3.7: The depths of the nodes in C17 considering worst case depths



Source: André Reis

Other data besides the depth of each AIG node that can be extracted by algorithms running on AIGs are, for example, the degree of each node, that is the number of trees subordinated to the node. Another example of data that can be extract is the degree of each tree, that is the degree of the node with the higher degree considering all nodes of the tree.

But for the purpose of performance evaluation of data structures, calculating the worst distance of each node from its reachable inputs will pass through all AIG nodes, which is enough for this evaluation.

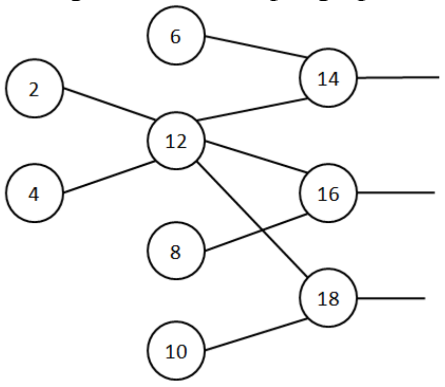
4 DIFFERENT DATA STRUCTURES FOR AND-INVERTER GRAPHS

The four implemented data structures for AIGs are discussed in this chapter. The goal is to describe the different data structures that are used in our experiments.

4.1 Overview of the data structures

The data structures for AIGs were implemented in four flavors. These are the combinations of two possibilities for two different characteristics discussed in the following subsections. The AIG in Fig. 4.1 will be used as an example in our discussion.

Figure 4.1: A simple graph.



Source: André Reis

4.1.1 Unidirectional vs Bidirectional

The information in an AIG is mainly unidirectional, in the sense that each AND node has the information about its two inputs, but each AND node does not have any information about the fan-out of the nodes.

However, the fan-out information can be discovered and stored in the AIG data structure. In this sense, an AIG can be unidirectional or bidirectional. In a unidirectional AIG, only the references for the inputs (fan-in) of the nodes are stored. In a bidirectional AIG, each node stores information about inputs (fan-in) and outputs (fan-out).

4.1.2 Pointer-based vs Index-based

An AIG data structure can also be pointer-based or index-based, considering the way it references neighbour nodes. When the reference is made through a pointer, the structure is said to be pointer-based. When the nodes are stored in a matrix and the index of the matrix is used to reference nodes the structure is said to be index-based.

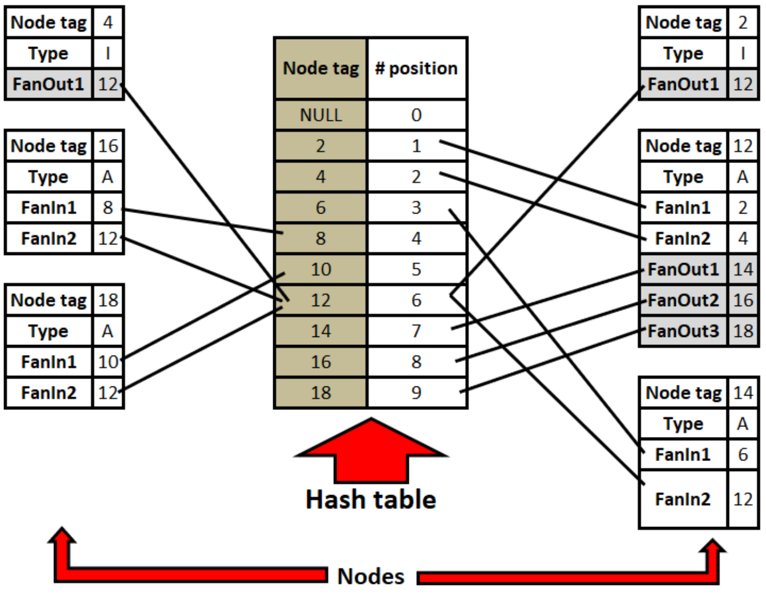
4.1.3 Four possible combinations

From the two characteristics discussed above, four combinations are possible. These are: 1) pointer-based bidirectional, 2) pointer-based unidirectional, 3) index-based bidirectional; and, 4) index-based unidirectional. In the following subsections, we discuss this four data structures in detail.

4.2 Pointer-based bidirectional data structure

This section presents a version of a pointer based bidirectional (backward and forward) data structure for AIGs. An illustration of the data structure can be seen in Fig. 4.2.

Figure 4.2: Pointer-based bidirectional data structure: all nodes are referenced as pointers



Source: Author

The data structure is composed of a hash table. This hash table is basically a vector of objects of type NodeBidirectional.

The class NodeBidirectional presents references to fan-in and fan-out nodes. This makes the data structure bidirectional. The references for fan-out and fan-in nodes are made through pointers, so the data structure is pointer-based. Consequently, this is a pointer-based bidirectional data structure for AIGs.

The number of fan-ins for each node is fixed in two because all AND nodes are two-input AND nodes. The number of fan-outs is not fixed because each node can be referenced in a fan-in of no node in the AIG, that is the case of output nodes.

In the case of AND nodes or input nodes the number of references will only be known in the moment that a new node is added to the graph and references it in its fan-in. To support this variation in the number of fan-outs, all nodes implement the property fan-out that is defined as a dynamic vector where a reference to a new node is added each time this node is included in the fan-in of a new node. When a new node is created, the number of fan-outs defined by default is zero, saving memory.

This vector with dynamic size definition give us flexibility to store any combinatorial circuit expressed by a benchmark like MTM (AMARú; GAILLARDON; MICHELI, 2015). If we were concerned only about synthetic benchmarks, that have a predictable number of fan-outs, we could implement the fan-outs as an array of fixed size.

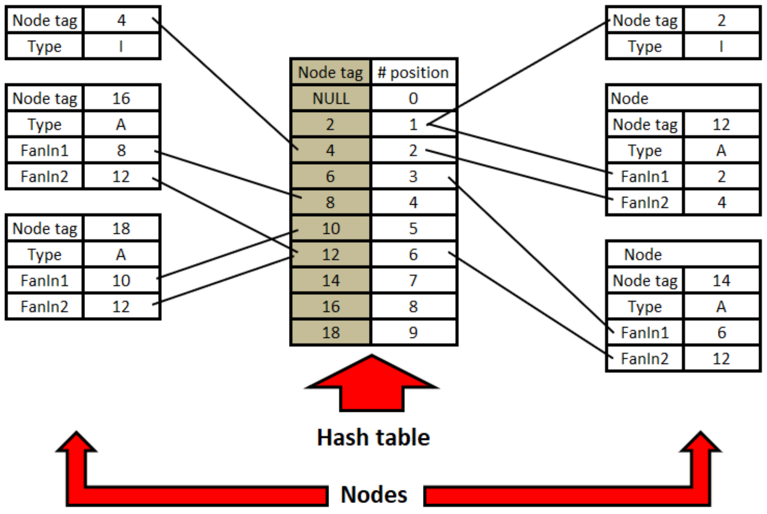
The use of a structure with dynamic size to store the fan-outs is crucial to save memory when storing nodes. Given that normally a pointer that references a new node will have 64-bits (4 bytes), if we pre-allocate 100 positions of fan-out for each node we will have 400 bytes for each node.

If we implement this fan-out pre-allocation approach using, for example, the sixteen MTM benchmark, were we have more than 16 million nodes, we will have 16 million multiplied by 400 bytes that will result in 64 billions bytes necesssary to store only the pre-allocated space to fan-outs. This corresponds to 64 gigabytes necessary only to pre-allocate the fixed size of fan-outs, that is impracticable. Using the dynamic allocation for each fan-out we will only allocate the necessary references to nodes that need to be connected to the fan-out of each node.

4.3 Pointer-based unidirectional data structure

This section presents a pointer-based unidirectional (backward) data structure for AIGs. An illustration of the data structure can be seen in Fig. 4.3.

Figure 4.3: Pointer-based unidirectional data structure: fan-outs are not implemented, only fan-ins.



Source: Author

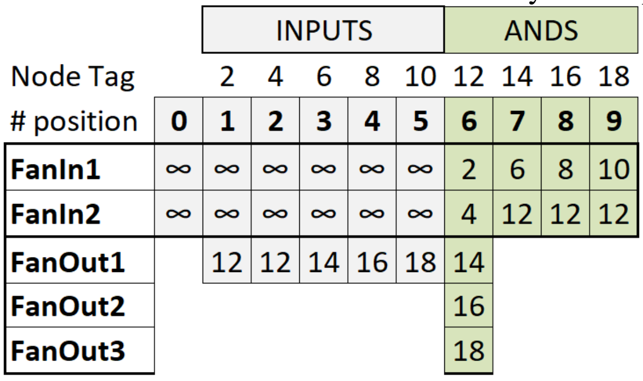
The data structure is composed of a hash table, whose is implemented as an array of node objects with the size equal to the number of nodes. This hash table is basically a vector of objects of type Node.

This properties of the class Node presents references only to fan-in nodes. This makes the data-structure unidirectional. The references for fan-in nodes are made through pointers, so the data structure is pointer-based. Consequently, this is a pointer-based bidirectional data structure for AIGs.

4.4 Index-based bidirectional data structure

This section presents an index-based bidirectional (backward and forward) data structure for AIGs. An illustration of the data structure can be seen in Fig. 4.4.

Figure 4.4: Index-based bidirectional data structure: array with inputs and ANDs



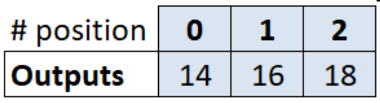
		INPUTS					ANDS			
Node Tag		2	4	6	8	10	12	14	16	18
# position	0	1	2	3	4	5	6	7	8	9
FanIn1	∞	∞	∞	∞	∞	∞	2	6	8	10
FanIn2	∞	∞	∞	∞	∞	∞	4	12	12	12
FanOut1		12	12	14	16	18	14			
FanOut2							16			
FanOut3							18			

Source: Author

The data structure is composed of two arrays of unsigned integers with fixed size reserved for fan-in1 and fan-in2 respectively. The size of each array is equal the number of nodes + 1. The fan-outs are stored in a dynamic matrix of unsigned integers with a dimension having the same size of the fixed size arrays and the other dimension having a dynamic size defined according to the number of fan-outs necessary for each AIG node, the same strategy used in the pointer-based bidirectional implementation to store fan-outs.

The data structure is complemented with an additional array that represents the output nodes. It is illustrated in Fig. 4.5.

Figure 4.5: Index-based data structure: array with outputs



# position	**0**	**1**	**2**
Outputs	14	16	18

Source: Author

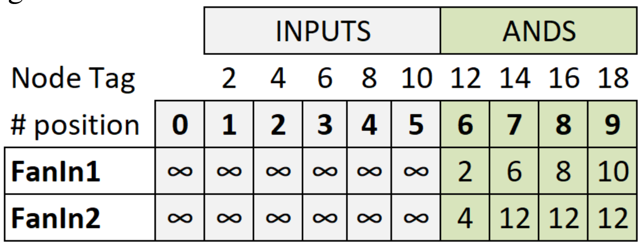
The array of output nodes is an array of unsigned integers. The size is equal the number of outputs.

The arrays arrayFanin1 and arrayFanIn2 present references to fan-in nodes. The dynamic matrix references to fan-out nodes. This makes the data-structure bidirectional. The references for fan-out and fan-in nodes are made throught unsigned integers that represent the index position in arrayFanIn1 and arrayFanIn2, so the data structure is index-based. Consequently, this is a index-based bidirectional data structure for AIGs.

4.5 Index-based unidirectional data structure

This section presents an index-based unidirectional (backward) data structure for AIGs. An illustration of the data structure can be seen in Fig. 4.6.

Figure 4.6: Index-based unidirectional data structure.



		INPUTS					ANDS			
Node Tag		2	4	6	8	10	12	14	16	18
# position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
FanIn1	∞	∞	∞	∞	∞	∞	2	6	8	10
FanIn2	∞	∞	∞	∞	∞	∞	4	12	12	12

Source: Author

The data structure is composed of two arrays of unsigned integers. The size is defined by the number of nodes + 1.

The data structure is complemented with an additional array that represents the output nodes. It is illustrated in Fig. 4.5, the same array output definition used in index-based bidirectional data structure.

The array of output nodes is an array of integers, the same array output definition used in index-based bidirectional data structure.

The arrays arrayFanIn1 and arrayFanIn2 present references to fan-in nodes. This makes the data-structure unidirectional. The references for fan-in nodes are made throught integers that represent the index position in arrayFanIn1 and arrayFanIn2, so the data structure is index-based. Consequently, this is a index-based unidirectional data structure for AIGs.

5 OPERATING SYSTEMS AND MEMORY MANAGEMENT

The work presented in this dissertation intends to measure the effect of different data structures on the efficient memory usage of a host machine by the operating system. In the previous chapter we discussed four alternative data structures to represent AIG graphs. In this chapter we discuss the memory hierarchy of current computers and how the operating system performs memory management.

5.1 Current computers

It is difficult to talk about current computers, as they keep always changing. For this reason we will maintain our discussion on a simplified level, just to introduce the concepts necessary to understand the measurements made in the scope of this dissertation. Current computers are based on a memory hierarchy, that is available on-chip. Typically, this means that the memory is organized in several hierarchical levels. This subject will be discussed in section 5.2. The access that programs have to the different memory levels is transparently managed by the operating system, as it will be highlighted in section 5.3. The efficiency that programs have to access the memory system through the operating system can be measured by monitoring tools. These tools will be discussed in sections 5.4, 5.5 and 5.6.

5.1.1 The computer used in the experiments

All experiments were carried out on server with:

- Processor: Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz;
- L1 cache: 4 x 32 KB 8-way set; associative instruction caches and 4 x 32 KB 8-way set associative data caches;
- L2 cache: 4 x 256 KB 4-way set associative caches;
- L3 cache: 8 MB 16-way set associative shared cache (CPU-WORLD, 2018), that is announced by Intel as 8 MB Smart Cache; [1] and
- RAM Memory: 64 GB.

[1]According to Intel(R) (INTEL, 2018): "Intel(R) Smart Cache refers to the architecture that allows all cores to dynamically share access to the last level cache."

Software running on server:

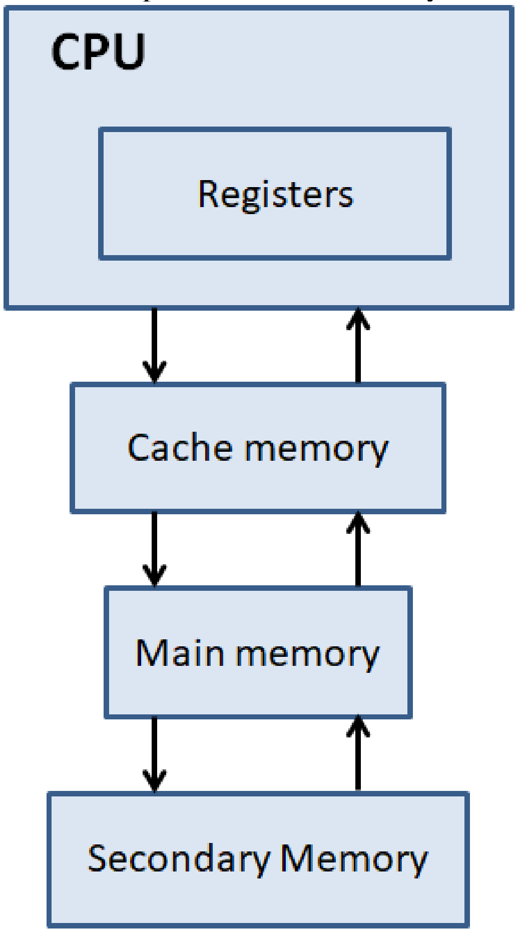
- Operating system: Linux 2.6.32-696.1.1.el6.x86_64;
- Compiler: g++ (GCC) 4.9.2 20150212 (Red Hat 4.9.2-6);
- PAPI version: 5.6.1.0; and
- Valgrind version: 3.8.1.

5.2 Memory Hierarchy

Ideally all data that the processor would need should be available at the same speed. But this is not possible considering the amount of data that is stored and needs to be available. As we have different types of memory that leads to different performance levels, the memory implementation strategy is to use a memory architecture composed as a hierarchy. The memory hierarchy has the goal of give to the processor all available memory in the lower level (cheapest) but with the speed of the highest (most expensive) memory (PATTERSON; HENNESSY, 2014). A simplified view of a memory hierarchy is presented in Fig. 5.1. Notice that the CPU cannot access directly all the memory levels of the memory hierarchy. For this reason, data must be exchanged between the memory levels and brought to a memory level in which the CPU can access them. So faster, more expensive and consequently smaller amount of memory is available near the processor. As far the distance from the processor, slower, cheaper and consequently bigger amount of memory is possible to be used.

Using a memory hierarchy, it is possible to provide a balanced performance using the speed of cache memory and the space with a low price per megabyte using hard disk and flash memory. The current computers implement multilevel caches, usually in three levels: L1, L2 and L3. The L1 cache level is designed for fast response. The L2 cache level is designed for reducing cache miss, avoiding to go to main memory, that is slower. Each core has its specific L1 and L2 cache. The L3 cache level is shared between all cores available, as we can see in section 5.1.1. In current processors, that is the case of the processor used in this experiment, the L1 cache is split in two caches with the same size where one part stores data and the other stores the instructions. L2 is normally unified, meaning that instructions and data are not separated. In the next section we discuss the implications of the associated memory management.

Figure 5.1: Simplified view of memory hierarchy.



Source: André Reis

5.3 Memory Management

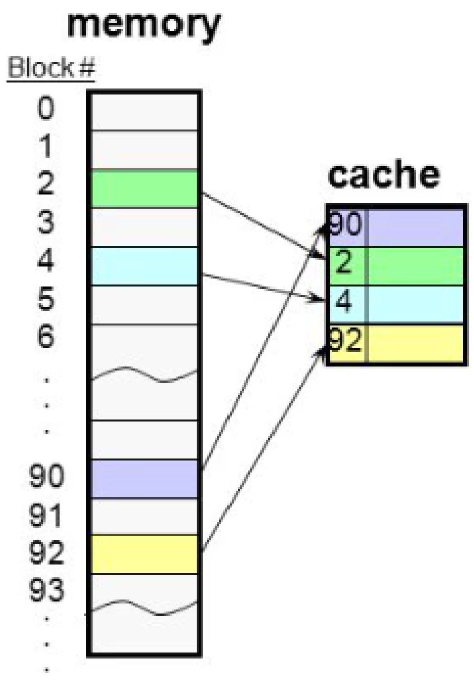
Due to the nature of the memory hierarchy, the main memory has a larger capacity than the cache memory. This concept is illustrated in Fig. 5.2. As a consequence, not all data addresses from the main memory are present in the cache memory. When a program tries to access a data that is not available in the cache memory, a cache miss occurs. At this moment the operating system has to transfer the data block from the main memory to the cache, so that it becomes accessible to the CPU. The block is the smallest portion of data that is copied from a lower level to an upper level. This block transfer implies in a delay for the program while it waits for the block to be transferred from main memory to the cache by the operating system.

When a cache miss occurs the time necessary to get the data is composed by the the memory penalty, that is the time spent to replace the data in the upper level with the block in the lower level. Other important concepts related to memory management are latency, known as response time, that is the total amount of time spent since the start and the end of an event, as for example to get data from the main memory. Another one is the bandwidth or throughput that is the total amount of work done in a period of time, for example the total of bytes transferred from the main memory (HENNESSY; PATTERSON, 2017). Locality influences the number of cache misses and it is categorized in temporal locality, that is when a data previously referenced has the probability to be referenced again in a short space of time and (PATTERSON; HENNESSY, 2014) spatial locality that deals with the high probability that another data in the same block will be used soon (HENNESSY; PATTERSON, 2017). The number of CPU clock cycles that CPU needs to wait from the memory, read or write, is called memory stall clock cycles. Regarding counting of cache misses, it is common to refer them as number of cache misses per 1000 instructions.

As write in cache is a operation that can delays the system, two different strategies can be used. It is called write through when data written in cache is written in the lower-level memory at the same time. It is called write back when the data is written in the lower-level of memory when the data is replaced in cache. The strategy is chosen based on the specific behavior of the software.

Regarding cache levels in different types of computers, in personal mobile devices, we usually have two cache levels (L1 and L2). Considering laptops, desktops and servers we have three cache levels (L1, L2 and L3). Regarding secondary memory, personal mobile devices use flash memory, laptops and desktops are converging to use flash memory and servers use part of flash memory and most of the part of hard disk (HENNESSY; PATTERSON, 2017).

Figure 5.2: Memory management maps main memory to cache.



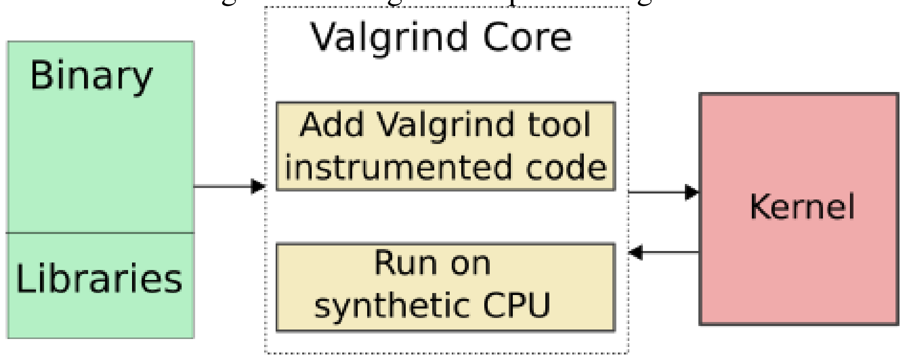
Source: André Reis

To monitor cache misses, current processors have counters that store the number of cache misses and another events defined as relevant by the processor manufacturer. Tools like Perf and PAPI use the data from these counters to show these information related to the parts of the program being executed. These two tools are commented in the subsequent sections, besides the Valgrind tool that works in a different way.

5.4 Valgrind Tool

Valgrind is better known as a suite of open source tools that serves to monitor memory management and profile programs in details but it is an instrumentation framework that is used to build dynamic analysis tools too (VALGRIND, 2018). These tools are designed to be as non-intrusive as possible. When running the program with Valgrind, it will take its control before it starts running it on a synthetic CPU provided by Valgrind core. Then it is directed to the selected tool that will add its specific instrumentation code to it and then the result will be send back to the core that will continue the execution of the instrumented code as we show in Fig. 5.3. So the programs to be analysed by Valgrind tools are dynamic binary instrumented and then they are run without the need of recompilation, unless you need additional information as lines of code that are associated to the error messages. If we want that Valgrind point out the error messages to the specific lines of code of the program, in C or C++, we can recompile our program and supporting libraries with -g option, enabling the debugging info. Another valuable option to use is -fno-inline that will list the function-call chain when we are working with C++ (VALGRIND, 2017).

Figure 5.3: Valgrind simplified diagram.



Source: Author

According to the official site (VALGRIND, 2018), Valgrind has six tools related to production-quality: a memory error detector (Memcheck), two thread error detectors (Helgrind and DRD), a call-graph generating cache and branch-prediction profiler (Cachegrind), and a heap profiler (Massif). There are three experimental tools included: a stack/global array overrun detector (SGCheck), a second heap profiler that examines how heap blocks are used (DHAT), and a SimPoint basic block vector generator (BBV). Valgrind tools run in a plethora of platforms such as x86, AMD64, PPC, MIPS an ARM supporting the operating systems Linux, Solaris, Darwin, Mac OS and Android.

In our work we used the default Valgrind tool aimed primarily at C and C++ programs called Memcheck that will monitor the memory usage of our software. It shows the memory peak when running the program in bytes and if the program has any memory leak. If any error is found, Memcheck informs immediately pointing out the source code line with the error because we enabled the debugging info when the program was compiled. To get this information the program runs 10 to 30 times slower than usual. When running experiments it is necessary to be aware of this performance penalty. Primarily we used Memcheck to check if the program is running without any unexpected behavior: accessing undesired memory areas, memory leaks and bad frees of head blocks. After that we monitored the memory usage of our software.

Inserting extra instrumentation code in the program that is being analyzed Memcheck checks all read and write of memory and intercepts calls to malloc/new/free/delete.

Executing these checks and intercepting these commands the tools are able to monitor commands that misuse the memory as trying to access not allowed memory areas, memory leaks and use of uninitialized variables (VALGRIND, 2018). We used Valgrind Memcheck tool to collect the memory usage for each running of the basic algorithm using the each of the four data structure implementations.

5.5 Perf Tool

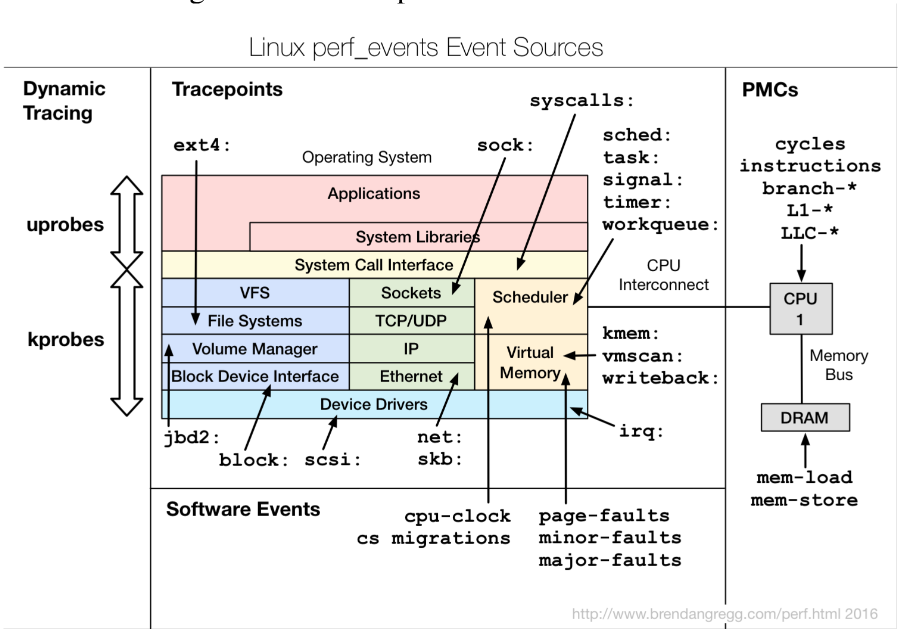
Perf is a Linux profiling tool with performance counter. Perf is implemented as a Linux command called perf and it is also called perf_events. It is included in the Linux kernel and so, it it the official Linux Performance profiler. It can instrument CPU performance counters, tracepoints, kprobes (kernel probes) and uprobes (known as dynamic tracing). Performance counters are special registers built inside modern processors that register hardware events as for example cache misses, or instructions executed. Fig. 5.4 shows the event sources collected by Perf. The availability of each counter is defined by each processor vendor. Perf can obtain event counts using the stat option, record events for later reporting using record option, or break down events by process using report option. (PERF, 2018). Its interface is very simple, it enables easily record the events selected to collect the results and generates a report with these results.

According to (GREGG, 2017) a Perf basic workflow can be defined as:

- list: find events
- stat: count them
- record: write event data to file
- report: browse summary
- script: event dump for post processing.

We tried to use Perf to collect the cache references and cache misses but given that we were not able to isolate the code running the basic path algorithm we opted out to use the PAPI tool, described in 5.6, where we were able to use this level of code isolation. The lack of code isolation will lead to read the total of performance counter when loading the input file into the memory, leading to data that will not be necessary to the experiment.

Figure 5.4: Linux perf-events Event Sources.



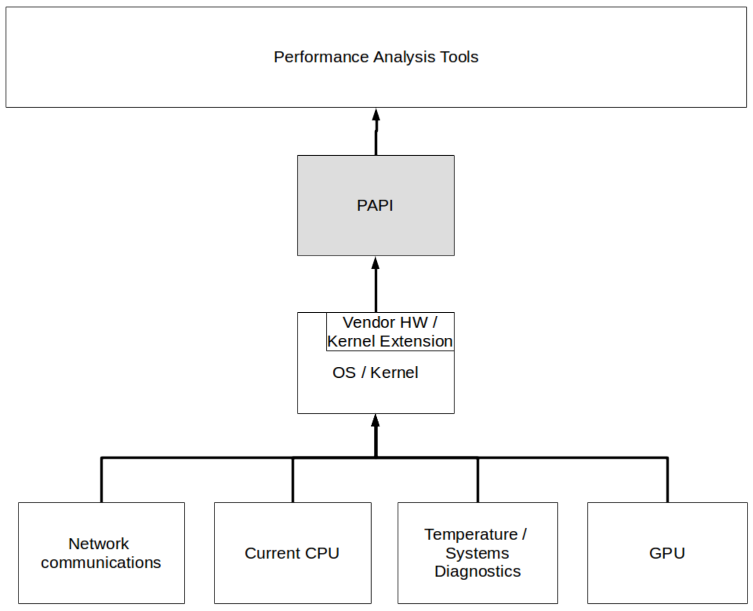
Source: Brendan Gregg's website (GREGG, 2018)

5.6 PAPI Tool

PAPI tool stands for Performance Application Programming Interface. According to the official website (PAPI, 2018), it lets to see the relationships between the software performance and hardware events when running the software. A lot of architectures are supported: AMD, ARM, CRAY, IBM, Infiniband, Intel, Lustre, NVIDIA and Virtual Environment. It has a low level and a high level interface.

PAPI is used in this work to collect the total number of data cache misses, specifically in L1 and L2 cache levels when running algorithm. Events of L3 data cache misses were not available for the processor used in this experiment. The cache misses are collect from hardware counters provided by PAPI API. Each architecture provides its set of available hardware counters and available events. In PAPI, we can check the available events using the command papi_avail. The Fig. 5.5 shows the flow of information to be received from PAPI. It collects information from the kernel and operating system.

Figure 5.5: PAPI tool flow of information.


Performance Analysis Tools
PAPI
Vendor HW / Kernel Extension
OS / Kernel
Network communications
Current CPU
Temperature / Systems Diagnostics
GPU

Source: Author adapted from PAPI official website

It was possible to collect the cache misses with this isolation using a wrapper available in the Open-Source Library for Physical Design Research and Teaching (GUTH; NETTO; LIVRAMENTO, 2018). It is important to note that to successfully capture the cache misses using PAPI, it is necessary to check if the host machine where the experiment will be run has the cache misses events available, the program must be compiled without optimization and the hardware counters must be reseted before running the experiment.

6 RESULTS

The work presented in this dissertation intends to show and analyze the performance when running a basic path algorithm using different data structures storing AIGs of different sizes. In the previous chapter we discussed the memory hierarchy of current computers and how the operating system performs memory management. In this chapter we present the results of the experiments with four different data structures regarding runtime, memory usage and cache misses.

During the runtime and memory usage test it was used the original compiled version of the four implementations. To run the experiment that collected the cache misses we inserted in the original code some instructions to collect the number of cache misses found only during the execution of the path algorithm. These instructions and the code used to call the PAPI API were obtained in Open-Source Library for Physical Design Research and Teaching (GUTH; NETTO; LIVRAMENTO, 2018). All results are expressed in figures of charts that relate all the four data structures with the specific criteria. To get all the criteria values we loaded synthetic AIGs with the number of nodes defined in function of the levels of nodes used.

Given that synthetic AIGs generated specifically for this work always have only one output and the number of inputs are defined according to the number of levels following the formula $I = 2^{(L-1)}$ where I is the number of inputs and L is the number of levels, these synthetic AIGs serve to the purpose of this work to exercise the four implementations using a controlled and crescent number of inputs, levels and number of ANDS. Considering that real combinational circuits with similar size normally do not have only one output and so many inputs, we included tests using benchmarks with more than one million nodes that are similar to real combinational circuits. These very large circuits are called more than million benchmarks (AMARú; GAILLARDON; MICHELI, 2015). They are available in binary AIGER format. These MTM benchmarks were designed to challenge the size capacity of modern optimization tools (AMARú; GAILLARDON; MICHELI, 2015). The information about the three MTM benchmarks used in the experiments of this work is presented in Table 6.1.

Notice in Table 6.1 that the largest benchmark, the twentythree, that has more than twenty three AND nodes has only 153 inputs and 68 outputs and a total of more than twenty three nodes with 176 levels. Our synthetic AIG with similar number of nodes, that is the AIG with 24 levels and a total of more than 16 millions of nodes, it has more than 8

Table 6.1: MTM Benchmark information. This table was adapted from (AMARú; GAILLARDON; MICHELI, 2015)

Name	Inputs	Outputs	AND nodes	Levels
sixteen	117	50	16216836	140
twenty	137	60	20732893	162
twentythree	153	68	23339737	176

million inputs and just one output. This example shows that the synthetic AIGs can serve for tests but are very different from real circuits. All nodes of synthetic AIGs were name tagged in depth order. The data was collected when running the basic path algorithm in forward direction (from inputs to outputs) registering the worst distance from all inputs until the output that is in listing 6.1.

The x-axis of all charts are the number of nodes processed. The points in the lines of each chart are marks that express a new level of AIGs. The mark at the rightest end of each line in the chart is the level 25. All tables and graphs that are related to the four implementations use abbreviations for each implementation, for example, IB for Index-based bidirectional data structure. All abbreviations are defined in the list of abbreviations and acronyms in the first pages of this work.

6.1 Basic path algorithm

To exercise the four data structure implementations we executed a basic path algorithm described in pseudo-code in Listing 6.1 that traverses all nodes using recursion registering the worst distance (higher distance) from each node from the inputs that can be reached by that node. Using recursion, the algorithm does not depend on the existence of bidirectional connection between the nodes. Then it can be used successfully in unidirectional and bidirectional implementations. This way, the depths from inputs to outputs can be computed with a recursive routine even if the information in both direction is missing.

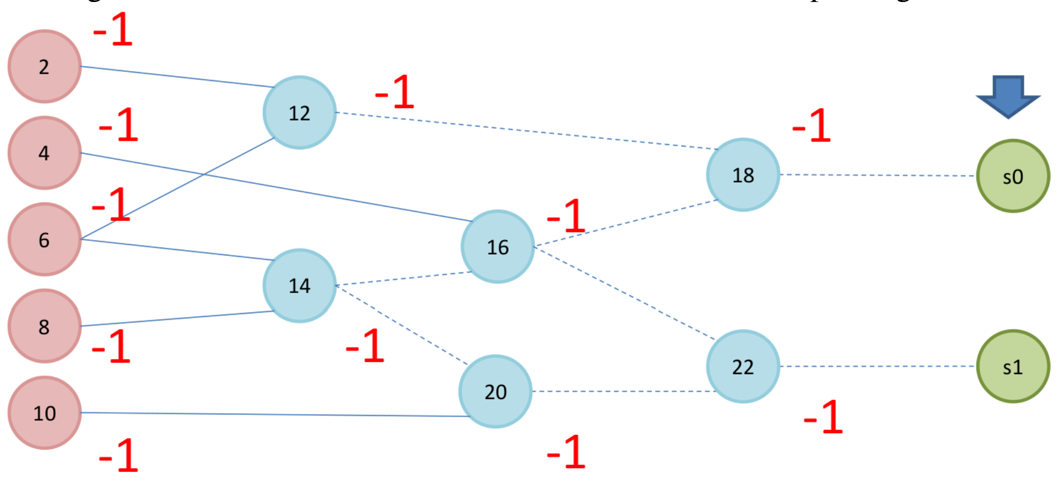
Listing 6.1 – Pseudo-code of path algorithm executed to test the four data structures.

```
1   int computeDepth(node n1) {
2       if (n1 is primary input)
3           return 0;
4       else{
5           int depthIn1= computeDepth (In1);
6           int depthIn0= computeDepth (In0);
7           return max(depthIn1, depthIn0)+1;
8       }
9   }
```

In lines 5 and 6 the recursion is called to compute the distance of the nodes in each fanIn, and so traversing the node in depth until reach the primary input, setting the distance in 0.

To illustrate how this algorithm works, we will demonstrate its execution in Table 6.2 using the AIG defined in Figure 6.1. Note that the difference from inputs for all nodes are set initially in the value of -1.

Figure 6.1: AIG used for demonstrate the execution of the path algorithm.



Source: André Reis

6.2 Runtime

The time consumed to execute the path algorithm along all the AIG is considered the runtime. To ensure the precision of time collected, the time started to be measured just before running path algorithm and the time measurement was stopped just before

Table 6.2: Algorithm execution showing computation number (Comp.), node being processed (Processing), updated distance (Dist.) and all the nodes distance after each computation.

Comp.	Processing	Dist.	2	4	6	8	10	12	14	16	18	20	22
1	s0	-	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	18	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
4	2	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	12	-1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	6	0	0	-1	0	-1	-1	-1	-1	-1	-1	-1	-1
7	12	1	0	-1	0	-1	-1	1	-1	-1	-1	-1	-1
8	18	-1	0	-1	0	-1	-1	1	-1	-1	-1	-1	-1
9	16	-1	0	-1	0	-1	-1	1	-1	-1	-1	-1	-1
10	4	0	0	0	0	-1	-1	1	-1	-1	-1	-1	-1
11	16	-1	0	0	0	-1	-1	1	-1	-1	-1	-1	-1
12	14	-1	0	0	0	-1	-1	1	-1	-1	-1	-1	-1
13	6	0	0	0	0	-1	-1	1	-1	-1	-1	-1	-1
14	14	-1	0	0	0	-1	-1	1	-1	-1	-1	-1	-1
15	8	0	0	0	0	0	-1	1	-1	-1	-1	-1	-1
16	14	1	0	0	0	0	-1	1	0	-1	-1	-1	-1
17	16	2	0	0	0	0	-1	1	0	2	-1	-1	-1
18	18	3	0	0	0	0	-1	1	0	2	3	-1	-1
19	s1	-	0	0	0	0	-1	1	0	2	3	-1	-1
20	22	-1	0	0	0	0	-1	1	0	2	3	-1	-1
21	16	2	0	0	0	0	-1	1	0	2	3	-1	-1
22	22	-1	0	0	0	0	-1	1	0	2	3	-1	-1
23	20	-1	0	0	0	0	-1	1	0	2	3	-1	-1
24	14	1	0	0	0	0	-1	1	1	2	3	-1	-1
25	20	-1	0	0	0	0	-1	1	1	2	3	-1	-1
26	10	0	0	0	0	0	0	1	1	2	3	-1	-1
27	20	2	0	0	0	0	0	1	1	2	3	2	-1
28	22	3	0	0	0	0	0	1	1	2	3	2	3

Table 6.3: Runtime of synthetic AIGs using each data structure implementation

Level	nodes	IB	PU	IU	PB
10	1,02E+03	5,00E-06	1,00E-05	7,00E-06	1,10E-05
11	2,05E+03	1,00E-05	1,90E-05	1,20E-05	2,00E-05
12	4,10E+03	1,90E-05	3,70E-05	2,30E-05	4,10E-05
13	8,19E+03	3,70E-05	7,40E-05	4,70E-05	8,70E-05
14	1,64E+04	7,40E-05	1,69E-04	9,00E-05	1,83E-04
15	3,28E+04	1,47E-04	3,04E-04	1,82E-04	4,22E-04
16	6,55E+04	2,96E-04	6,24E-04	3,61E-04	9,06E-04
17	1,31E+05	5,91E-04	1,27E-03	7,20E-04	1,83E-03
18	2,62E+05	1,20E-03	2,61E-03	1,44E-03	3,64E-03
19	5,24E+05	2,41E-03	5,29E-03	3,10E-03	7,31E-03
20	1,05E+06	4,75E-03	1,05E-02	5,80E-03	1,46E-02
21	2,10E+06	9,51E-03	2,13E-02	1,16E-02	2,95E-02
22	4,19E+06	1,91E-02	4,22E-02	2,33E-02	5,86E-02
23	8,39E+06	3,93E-02	8,46E-02	4,63E-02	1,17E-01
24	1,68E+07	7,58E-02	1,69E-01	9,28E-02	2,36E-01
25	3,36E+07	1,57E-01	3,38E-01	1,87E-01	4,68E-01

Table 6.4: Runtime of MTM Benchmarks using each data structure implementation

Benchmark	nodes	IB	PU	IU	PB
sixteen	1,62E+07	1,04E+00	1,41E+00	1,47E+00	1,59E+00
twenty	2,07E+07	1,48E+00	1,95E+00	2,05E+00	2,22E+00
twentythree	2,33E+07	1,72E+00	2,28E+00	2,36E+00	2,57E+00

finishing the execution of path algorithm. This was done directly in the source code of the software that implements each data structure and has the path algorithm.

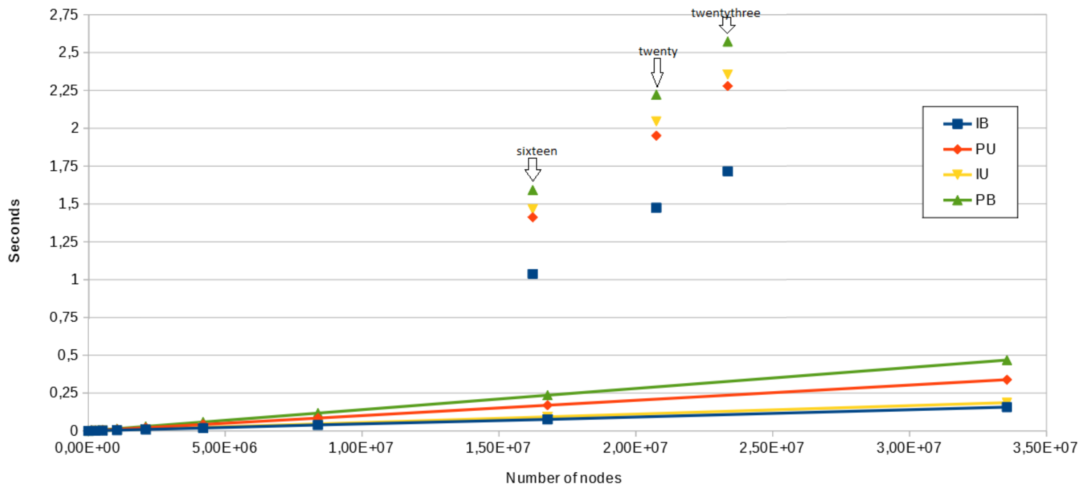
All further analyzed data of the runtime of synthetic AIGs are presented in the Table 6.3. The Table 6.4 presents the runtime of the MTM Benchmarks.

Regarding synthetic AIGs we can see in Fig. 6.2 that the runtime for index-based unidirectional is higher than index-based bidirectional. The average difference is 23.18%. This is an unexpected result because we expected that index-based unidirectional implementation had the best runtime considering that this is the smaller implementation and so we would have less data to process when running the path algorithm. Considering the both pointer-based implementation, the result occurred as expected with the smaller implementation in terms of node size having the best performance, the pointer-based unidirectional.

The implementation with best performance and the one with worst performance for synthetic AIGs, respectively the index-based bidirectional and the pointer-based bidirectional, are the same when considering the MTM Benchmarks. The results are different when comparing the runtime of pointer-based unidirectional and index-based unidirectional, where the performance of the first implementation is better than the second one.

This difference in the positions of the implementations when comparing the synthetic benchmarks and the MTM benchmarks shows an opportunity for further investigation using different algorithms and even different combinational circuits trying to identify what are the causes of this differences and if this is a pattern for every combinational circuit with features different from synthetic benchmarks.

Figure 6.2: Runtime of running path algorithm in all four data structures.



Source: Author

6.3 Memory usage

All the data was collected using the Valgrind tool Memcheck (VALGRIND, 2018) using the command listed in Listing 6.2.

Listing 6.2 – Valgrind command used to collect memory usage.

```
valgrind ./<program> <program parameters>
```

It is important to note that it was not possible to isolate the code that executed the path algorithm when collecting memory usage given that Valgrind tool Memcheck executes the program as it was originally compiled. Then, the results of memory usage consider the total memory usage along all the execution of each implementation since the reading of each node defined in the AIGER file and the storing in each position of the respective implementation.

Considering that the memory usage is proportional to the number of nodes being

Table 6.5: Memory usage for synthetic AIGs using each data structure implementation

Level	nodes	IB	PU	IU	PB
10	1,02E+03	7,24E+04	8,07E+04	4,37E+04	1,63E+05
11	2,05E+03	1,13E+05	1,30E+05	5,60E+04	2,94E+05
12	4,10E+03	1,95E+05	2,28E+05	8,06E+04	5,56E+05
13	8,19E+03	3,59E+05	4,25E+05	1,30E+05	1,08E+06
14	1,64E+04	6,87E+05	8,18E+05	2,28E+05	2,13E+06
15	3,28E+04	1,34E+06	1,60E+06	4,25E+05	4,23E+06
16	6,55E+04	2,65E+06	3,18E+06	8,18E+05	8,42E+06
17	1,31E+05	5,27E+06	6,32E+06	1,60E+06	1,68E+07
18	2,62E+05	1,05E+07	1,26E+07	3,18E+06	3,36E+07
19	5,24E+05	2,10E+07	2,52E+07	6,32E+06	6,71E+07
20	1,05E+06	4,20E+07	5,04E+07	1,26E+07	1,34E+08
21	2,10E+06	8,39E+07	1,01E+08	2,52E+07	2,68E+08
22	4,19E+06	1,68E+08	2,01E+08	5,04E+07	5,37E+08
23	8,39E+06	3,36E+08	4,03E+08	1,01E+08	1,07E+09
24	1,68E+07	6,71E+08	8,05E+08	2,01E+08	2,15E+09
25	3,36E+07	1,34E+09	1,61E+09	4,03E+08	4,29E+09

Table 6.6: Memory usage for MTM Benchmarks using each data structure implementation

Benchmark	nodes	IB	PU	IU	PB
sixteen	1,62E+07	8,20E+08	7,78E+08	1,95E+08	2,42E+09
twenty	2,07E+07	1,09E+09	9,95E+08	2,49E+08	3,17E+09
twentythree	2,33E+07	1,22E+09	1,12E+09	2,80E+08	3,57E+09

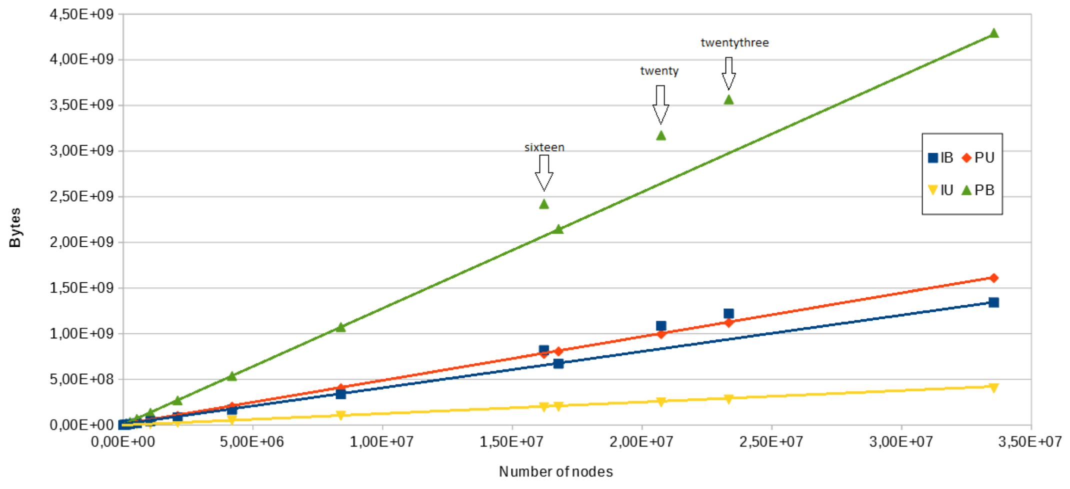
processed, we believe that this incapacity of code isolation to get the memory usage only when running the path algorithm does not invalidate the experiments.

All further analyzed data of the memory usage of synthetic AIGs are presented in the Table 6.5. The Table 6.6 presents the memory usage of the MTM Benchmarks.

Considering synthetic benchmarks, we got the expected results in the order of the best performance (less memory usage) to the worst performance of each implementation: index-based unidirectional, index-based bidirectional, pointer-based unidirectional and pointer-based bidirectional. Given that we have a linear increase in the memory usage according to the number of nodes, we believe that these positions in the ranking of memory usage reflect a trivial conclusion: smaller node in size occupy less memory.

Regarding MTM Benchmarks, the difference in memory usage compared to synthetic benchmarks is that index-based bidirectional implementation has a sensible higher memory usage than pointer-based unidirectional. As it was noticed with runtime, this difference can be explored in future works related to AIG data structure implementations.

Figure 6.3: Memory usage after running path algorithm in all four data structures.



Source: Author

6.4 Cache misses

Only cache misses that occurred running the path algorithm in each of the four data structures will be considered in our results. It was tried to get all the cache references and all the cache misses aiming to find out the percentage of cache misses according to all cache references using the Perf tool, but when running the command to get all the cache references and cache misses, it was noticed that the variation running the same command was higher than 30%. It was not possible to isolate the code that runs the path algorithm too. So we decided not to use the Perf tool to collect cache misses data because this high percentage of variation would invalidate the isolation necessary to use the cache reference and cache miss as a valid measure.

So another tool was used, the PAPI tool (PAPI, 2018). Using this tool we were not able to collect the percentage of cache misses considering all cache references, but we were able to collect the cache misses that occurred in the isolated code that runs the path algorithm. We were able to collect cache misses for cache levels 1 and 2. The PAPI events for cache misses in cache level 3 was not available. In PAPI Preset Events list cache level 3 is called PAPI_L3_DCM (Level 3 Data Cache Misses) after running the papi_avail command in the host computer.

We measured the L1 and L2 cache level events, specifically data cache misses. The results for synthetic benchmark for each respective cache level is presented in Table 6.7 and Table 6.9. The results for MTM benchmarks are showed in Table 6.8 and Table 6.10. PAPI Preset Events for cache misses in level 1 and level 2 are called respectively

PAPI_L1_DCM (Level 1 Data Cache Misses) and PAPI_L2_DCM (Level 2 Data Cache Misses).

Considering the synthetic benchmarks, despite the different cache level (L2), and different quantity of cache misses as it can be seen in Fig. 6.5, the scenario is the same in L1 cache as it can be seen in Fig. 6.4: the data structure with less cache misses is index-based bidirectional. Still considering only the synthetic benchmarks, it is important to note that in the graph in Fig. 6.4 and Fig. 6.5 the number of cache misses of index-based unidirectional is almost equal to the respective number of cache misses of index-based bidirectional. So, in these both graphs, the line for index-based unidirectional is hidden behind the line for index-based bidirectional. In L1 cache level, the number of cache misses of index-based unidirectional is in average 1.05% higher than index-based bidirectional. In L2 cache level, the position of number of cache misses for the both index-based implementations is the same but the difference is a little higher than in the L1 cache level, 1,4%.

Regarding MTM Benchmarks we have the same result found in runtime and memory usage tests: the ranking of implementations is different when compared to synthetic benchmarks. Pointer-based unidirectional presented the lower number of cache misses followed by index-based bidirectional, index-based unidirectional and then pointer-based bidirectional. Comparing the results from L1 and L2 cache levels for MTM Benchmarks, we can see that in L2 cache level, the difference between pointer-based unidirectional and index-based bidirectional is lower than in L1 cache level. On the other hand, in L1 cache level the difference between index-based bidirectional, index-based unidirectional and pointer-based bidirectional is lower when compared to the same three implementation in L2 cache level.

Table 6.7: L1 cache misses for synthetic AIGs using each data structure implementation

Level	nodes	IB	PU	IU	PB
10	1,02E+03	1,60E+02	8,18E+02	1,67E+02	1,62E+03
11	2,05E+03	2,91E+02	1,60E+03	2,90E+02	3,20E+03
12	4,10E+03	5,67E+02	3,15E+03	5,62E+02	6,37E+03
13	8,19E+03	1,09E+03	6,26E+03	1,09E+03	1,27E+04
14	1,64E+04	2,14E+03	1,25E+04	2,14E+03	2,53E+04
15	3,28E+04	4,28E+03	2,48E+04	4,27E+03	5,05E+04
16	6,55E+04	8,52E+03	4,96E+04	8,56E+03	1,01E+05
17	1,31E+05	1,70E+04	9,90E+04	1,69E+04	2,01E+05
18	2,62E+05	3,39E+04	1,98E+05	3,39E+04	4,04E+05
19	5,24E+05	6,74E+04	3,96E+05	6,76E+04	8,05E+05
20	1,05E+06	1,35E+05	7,92E+05	1,35E+05	1,61E+06
21	2,10E+06	2,69E+05	1,58E+06	2,73E+05	3,23E+06
22	4,19E+06	5,36E+05	3,18E+06	5,37E+05	6,45E+06
23	8,39E+06	1,07E+06	6,33E+06	1,07E+06	1,29E+07
24	1,68E+07	2,14E+06	1,27E+07	2,14E+06	2,58E+07
25	3,36E+07	4,29E+06	2,54E+07	4,38E+06	5,16E+07

Table 6.8: L1 cache misses for MTM Benchmarks using each data structure implementation

Benchmark	nodes	IB	PU	IU	PB
sixteen	1,62E+07	7,23E+07	5,43E+07	7,31E+07	7,43E+07
twenty	2,07E+07	1,03E+08	7,93E+07	1,04E+08	1,08E+08
twentythree	2,33E+07	1,20E+08	9,16E+07	1,22E+08	1,26E+08

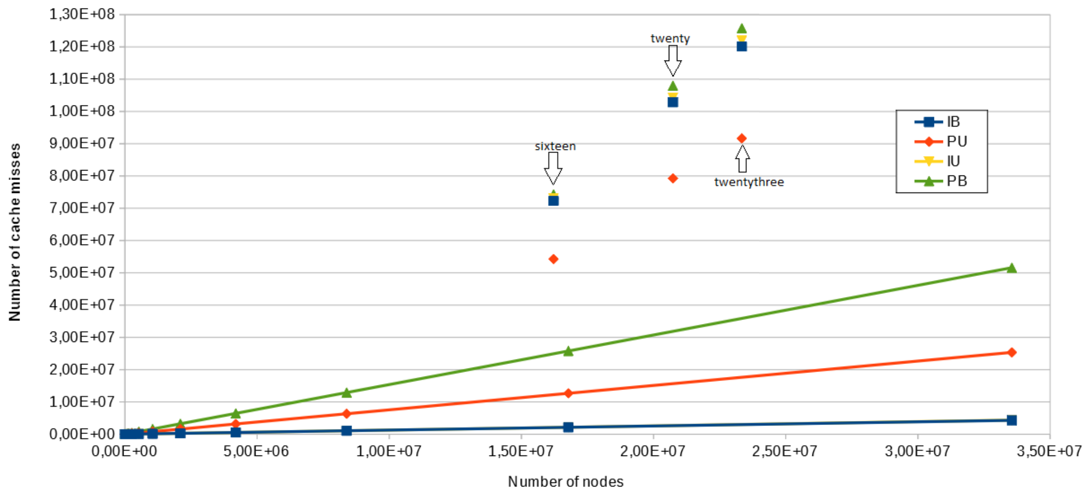
Table 6.9: L2 cache misses for synthetic AIGs using each data structure implementation

Level	nodes	IB	PU	IU	PB
10	1,02E+03	4,93E+02	1,69E+03	5,09E+02	3,89E+03
11	2,05E+03	7,41E+02	3,37E+03	7,37E+02	7,49E+03
12	4,10E+03	1,34E+03	6,43E+03	1,29E+03	1,48E+04
13	8,19E+03	2,44E+03	1,26E+04	2,45E+03	2,96E+04
14	1,64E+04	4,74E+03	2,51E+04	4,87E+03	5,81E+04
15	3,28E+04	9,42E+03	5,01E+04	9,62E+03	1,16E+05
16	6,55E+04	1,88E+04	1,02E+05	1,90E+04	2,31E+05
17	1,31E+05	3,73E+04	1,98E+05	3,76E+04	4,61E+05
18	2,62E+05	7,40E+04	3,95E+05	7,40E+04	9,21E+05
19	5,24E+05	1,48E+05	7,91E+05	1,48E+05	1,84E+06
20	1,05E+06	2,95E+05	1,58E+06	2,94E+05	3,68E+06
21	2,10E+06	5,88E+05	3,17E+06	5,92E+05	7,37E+06
22	4,19E+06	1,17E+06	6,33E+06	1,18E+06	1,48E+07
23	8,39E+06	2,35E+06	1,27E+07	2,35E+06	2,95E+07
24	1,68E+07	4,67E+06	2,53E+07	4,69E+06	5,89E+07
25	3,36E+07	9,40E+06	5,06E+07	9,47E+06	1,18E+08

Table 6.10: L2 cache misses for MTM Benchmarks using each data structure implementation

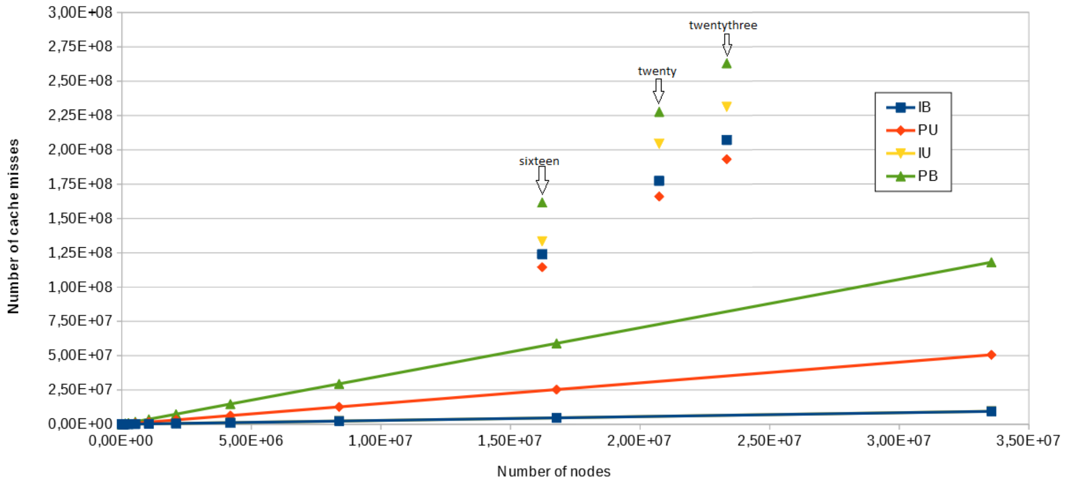
Benchmark	nodes	IB	PU	IU	PB
sixteen	1,62E+07	1,24E+08	1,15E+08	1,33E+08	1,62E+08
twenty	2,07E+07	1,77E+08	1,66E+08	2,04E+08	2,28E+08
twentythree	2,33E+07	2,07E+08	1,93E+08	2,31E+08	2,63E+08

Figure 6.4: L1 Data cache misses count after running path algorithm in all four data structures.



Source: Author

Figure 6.5: L2 Data cache misses count after running path algorithm in all four data structures.



Source: Author

This chapter has presented the experiment results that collected runtime, memory usage and cache misses when running the path algorithm that calculates the distance nodes using the four data structures. These results showed the behavior of each data structure with synthetic AIGs with 10 to 25 levels of nodes and with MTM Benchmarks. It was showed both in the results of synthetic benchmarks and MTM benchmarks that index-based bidirectional data structure is more efficient in term of time as it can be seen in Fig. 6.2 compared to the respective unidirectional data structure. But there is a price to pay: it is necessary to store the fanouts, which implies in store more data to have more runtime efficiency, as we can see in Fig. 6.3 where the index-based bidirectional is the second data structure that presented more memory usage. Considering the balance of runtime, memory usage and cache misses of synthetic Benchmarks and MTM Benchmarks, index-based bidirectional data structure is the best choice comparing to the other three data structures.

7 CONCLUSION

In this work we have implemented and compared four different data structures for And-Inverter Graphs. Our goal was to verify in practice that the implementation of AIGs as an integer index-based unidirectional data structure presents in practice a superior performance compared to other implementations. This has been verified through the results presented in the previous chapter. We believe this superior performance comes from three characteristics. First, using integer indexes instead of pointers results in a more compact data structure, because integers have 32 bits, while pointers have 64 bits. Using integer indexes results then in a more compact data structure. Second, using a data structure that is unidirectional also reduces the memory consumption, as references are stored in a single direction (backwards). Sometimes this requires some creativity, to implement algorithms with unidirectional references. Third, using vectors (instead of pointers) results in more memory locality, as vector elements are stored contiguously. This conclusion is valid to synthetic benchmarks.

Considering only MTM benchmarks performance results, the pointer-based unidirectional showed the best results. This suggests that when using real circuits the use of pointer-based implementation can be a good approach. But it is necessary further studies to conclude this regarding MTM benchmarks.

These are important conclusions, but still they have to be taken cautiously due to some limitations in the scope of this work. These limitations are discussed in the next section.

7.1 Limitations of this work

This work has some important limitations that must be highlighted. In the following we identify these limitations for future readers to understand these limitations and perhaps improve the experiments.

First, we have chosen only four different implementations of data structures that serve for the purpose of the evaluation of the impacts of pointer and pointerless strategy. Given that we got different results when testing using synthetic benchmarks and using MTM benchmarks it would be important to measure the behavior of different data structures for circuits such as adjacency matrix, incidence matrix and adjacency list. These expanded experiments could help to understand what is the best data structure to fit each different need.

Second, we tested for a single algorithm. A more thorough analysis would be needed to test using different algorithms. Some suggestion of different algorithms are BFS-based algorithms given that our implemented algorithm is a DFS-based algorithm, or for example the well-known Dijkstra algorithm. However, there is a considered amount of work involved in this experiment, as it would be necessary to implement each algorithm four times, one for each data structure.

Third, the use of a vector to implement the hash table in the pointer-based data structure gives a vectorial characteristic to the pointer-based data structures. In this case, the data is accessed by a pointer reference, but all pointer references are allocated in a vector that implements sequential organization. A purely pointer based data structure would be more sparse in nature and could be a better way of implementation to test the impact of the use of a data structure completely implemented that is not oriented to data locality. We believe that using a purely pointer-based data structure would led to worst results in runtime and cache misses tests.

7.2 Final Remarks and Future Work

Despite the limitations cited, this work is a first step towards evaluating different data structures for AIGs. We have been careful to highlight these limitations so that future work can consider these points and develop a more complete evaluation. Besides the suggestions already mentioned in the previous section, here we list more suggestions that could expand this work.

The experiments were executed in a multi-core CPU but this architecture was not explored in level of multi-threading in the software implementation. A suggestion of future work would be to explore the multi-core architecture in the software implementation to understand what could be the possible gains and losses. A work in this direction related to BDD was published by Elbayoumi, Hsiao and ElNainay (ELBAYOUMI; HSIAO; ELNAINAY, 2013) that could be an inspiration to research in the same direction, but using AIGs.

Machine learning is an area of expertise that could be explored to construct algorithms that could decide the best data structure to be used based on the circuit structure and future application of this circuit. Beerel and Pedram (BEEREL; PEDRAM, 2018) present opportunities for machine learning in EDA.

REFERENCES

AMARú, L.; GAILLARDON, P.-E.; MICHELI, G. D. The epfl combinational benchmark suite. In: **INT'L WORKSHOP ON LOGIC AND SYNTHESIS (IWLS). Proceedings**. Mountain View, California, USA: [s.n.], 2015. p. 5. Available from Internet: <https://infoscience.epfl.ch/record/207551>.

AMMANN, P.; OFFUTT, J. Input space partitioning. In: **Introduction to software testing**. 2. ed. New York, NY, USA: Cambridge University Press, 2017. chp. 6, p. 75–105.

BEEREL, P. A.; PEDRAM, M. Opportunities for machine learning in electronic design automation. In: **INT'L SYMPOSIUM ON CIRCUITS AND SYSTEMS (ISCAS). Proceedings**. [S.l.]: IEEE, 2018. p. 1–5. ISSN 2379-447X.

BEM, V. D. et al. Impact and optimization of lithography-aware regular layout in digital circuit design. In: IEEE. **INT'L CONFERENCE ON COMPUTER DESIGN (ICCD). Proceedings**. [S.l.], 2011. p. 279–284.

Berkeley Logic Synthesis and Verification Group. **ABC: A System for Sequential Synthesis and Verification**. 2018. Available from Internet: <http://www.eecs.berkeley.edu/~alanmi/abc/>.

BIERE, A. **The AIGER And-Inverter Graph (AIG) Format Version 20071012**. Linz, Austria: [s.n.], 2007. Available from Internet: <http://fmv.jku.at/papers/Biere-FMV-TR-07-1.pdf>.

BRAYTON, R.; MISHCHENKO, A. Abc: An academic industrial-strength verification tool. In: **COMPUTER AIDED VERIFICATION (CAV). Proceedings**. Edinburgh, UK: Springer, 2010. p. 24–40. Available from Internet: <http://dx.doi.org/10.1007/978-3-642-14295-6_5>.

BUTZEN, P. F. et al. Transistor network restructuring against nbti degradation. **Microelectronics Reliability**, Pergamon, v. 50, n. 9, p. 1298–1303, 2010. Available from Internet: <https://doi.org/10.1016/j.microrel.2010.07.140>.

BUTZEN, P. F. et al. Design of cmos logic gates with enhanced robustness against aging degradation. **Microelectronics Reliability**, Pergamon, v. 52, n. 9-10, p. 1822–1826, 2012.

BUTZEN, P. F. et al. Standby power consumption estimation by interacting leakage current mechanisms in nanoscaled cmos digital circuits. **Microelectronics Journal**, Elsevier, v. 41, n. 4, p. 247–255, 2010.

BUTZEN, P. F. et al. Modeling and estimating leakage current in series-parallel cmos networks. In: **GREAT LAKES SYMPOSIUM ON VLSI (GLSVLSI). Proceedings**. ACM, 2007. p. 269–274. Available from Internet: <http://doi.acm.org/10.1145/1228784.1228852>.

CALLEGARO, V. et al. Read-polarity-once boolean functions. In: **SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI). Proceedings**. IEEE, 2013. p. 1–6. Available from Internet: <https://doi.org/10.1109/SBCCI.2013.6644862>.

CORREIA, V.; REIS, A. Advanced technology mapping for standard-cell generators. In: **SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN. Proceedings**. IEEE, 2004. p. 254–259. Available from Internet: <https://doi.org/10.1109/SBCCI.2004.240925>.

CORREIA, V. P.; REIS, A. I. Classifying n-input boolean functions. In: **WORKSHOP IBERCHIP. Proceedings**. [S.l.: s.n.], 2001. p. 58.

CPU-WORLD. **Intel Core i7-7700K specifications**. 2018. Available from Internet: <http://www.cpu-world.com/CPUs/Core_i7/Intel-Core\%20i7\%20i7-7700K.html>.

ELBAYOUMI, M.; HSIAO, M. S.; ELNAINAY, M. A novel concurrent cache-friendly binary decision diagram construction for multi-core platforms. In: **DESIGN, AUTOMATION & TEST IN EUROPE (DATE). Proceedings**. Grenoble, France: [s.n.], 2013. p. 4. ISSN 1530-1591. Available from Internet: <https://doi.org/10.7873/DATE.2013.291>.

FIGUEIRÓ, T.; RIBAS, R. P.; REIS, A. I. Constructive aig optimization considering input weights. In: **INT'L SYMPOSIUM ON QUALITY ELECTRONIC DESIGN (ISQED). Proceedings**. Santa Clara, CA, USA: IEEE, 2011. p. 1–8.

FIšER, P.; SCHMIDT, J.; BALCáREK, J. Sources of bias in eda tools and its influence. In: **INT'L SYMPOSIUM ON DESIGN AND DIAGNOSTICS OF ELECTRONIC CIRCUITS SYSTEMS (DDECS). Proceedings**. Warsaw, Poland: IEEE, 2014. p. 258–261. Available from Internet: <https://www.researchgate.net/publication/269272311_Sources_of_bias_in_EDA_tools_and_its_influence>.

FONTANA, T. et al. How game engines can inspire eda tools development: A use case for an open-source physical design library. In: **INT'L SYMPOSIUM ON PHYSICAL DESIGN (ISPD). Proceedings**. Portland, Oregon, USA: ACM, 2017. p. 25–31. Available from Internet: <http://doi.acm.org/10.1145/3036669.3038248>.

FONTANA, T. A. et al. Exploiting cache locality to speedup register clustering. In: **SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI). Proceedings**. Fortaleza, Ceará, Brazil: ACM, 2017. p. 191–197. Available from Internet: <http://doi.acm.org/10.1145/3109984.3110005>.

GOMES, I. et al. Using only redundant modules with approximate logic to reduce drastically area overhead in tmr. In: **LATIN-AMERICAN TEST SYMPOSIUM (LATS). Proceedings**. [S.l.]: IEEE, 2015. p. 1–6.

GOMES, I. A. et al. Methodology for achieving best trade-off of area and fault masking coverage in atmr. In: **LATIN AMERICAN TEST WORKSHOP (LATW). Proceedings**. [S.l.]: IEEE, 2014. p. 1–6.

GREGG, B. **Kernel Recipes 2017 - Perf in Netflix**. 2017. Available from Internet: <https://www.youtube.com/watch?time_continue=771&v=UVM3WX8Lq2k>.

GREGG, B. **Perf Examples**. 2018. Available from Internet: <http://www.brendangregg.com/perf.html>.

GUTH, C.; NETTO, R.; LIVRAMENTO, V. **Open-Source Library for Physical Design Research and Teaching**. Santa Catarina, Brazil: [s.n.], 2018. Available from Internet: <https://github.com/TiagoAFontana/ophidian>.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture: A Quantitative Approach**. 6. ed. [S.l.]: Morgan Kaufmann, 2017. 936 p. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 978-0128119051.

INTEL. **Intel(R) Core(TM) i7-7700K Processor web page**. 2018. Available from Internet: <https://www.intel.com/content/www/us/en/products/processors/core/i7-processors/i7-7700k.html>.

JANSSEN, G. Design of a pointerless bdd package. In: **INT'L WORKSHOP ON LOGIC AND SYNTHESIS (IWLS). Proceedings**. [s.n.], 2001. Available from Internet: <https://www.research.ibm.com/haifa/projects/verification/SixthSense/papers/bdd_iwls_01.pdf>.

JANSSEN, G. A consumer report on bdd packages. In: **SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI). Proceedings**. Sao Paulo, Brazil: IEEE, 2003. p. 217–222. Available from Internet: <https://doi.org/10.1109/SBCCI.2003.1232832>.

JUNIOR, L. S. d. R. **Automatic generation and evaluation of transistor networks in different logic styles**. Thesis (PhD) — Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil, 2008.

JUNIOR, L. S. d. R. et al. A comparative study of cmos gates with minimum transistor stacks. In: **CONFERENCE ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI). Proceedings**. [S.l.]: ACM, 2007. p. 93–98.

JUNIOR, L. S. da R. et al. Fast disjoint transistor networks from bdds. In: **SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN. Proceedings**. [S.l.]: ACM, 2006. p. 137–142.

KLOCK, C. E. et al. Karma: a didactic tool for two-level logic synthesis. In: **INT'L CONFERENCE ON MICROELECTRONIC SYSTEMS EDUCATION (MSE). Proceedings**. [S.l.]: IEEE, 2007. p. 59–60.

LogiCS Research Lab. Simple Flow. In: . [s.n.], 2017. Release 20170710. Available from Internet: <http://www.inf.ufrgs.br/logics/downloads/>.

MACHADO, L. et al. Kl-cut based digital circuit remapping. In: **NORCHIP. Proceedings**. [S.l.]: IEEE, 2012. p. 1–4.

MANNE, S.; GRUNWALD, D.; SOMENZI, F. Remembrance of things past: Locality and memory in bdds. In: **DESIGN AUTOMATION CONFERENCE (DAC). Proceedings**. Anaheim, CA, USA: IEEE, 1997. p. 196–201. Available from Internet: <https://doi.org/10.1109/DAC.1997.597143>.

MARQUES, F. et al. Dag based library-free technology mapping. In: **GREAT LAKES SYMPOSIUM ON VLSI. Proceedings**. [S.l.]: ACM, 2007. p. 293–298.

MARQUES, F. S. et al. A new approach to the use of satisfiability in false path detection. In: **GREAT LAKES SYMPOSIUM ON VLSI. Proceedings**. [S.l.]: ACM, 2005. p. 308–311.

MARRANGHELLO, F. S. et al. Factored forms for memristive material implication stateful logic. **Journal on Emerging and Selected Topics in Circuits and Systems**, IEEE, v. 5, n. 2, p. 267–278, 2015.

MARTINELLO, O. et al. Kl-cuts: a new approach for logic synthesis targeting multiple output blocks. In: **DESIGN, AUTOMATION & TEST IN EUROPE (DATE). Proceedings**. [S.l.]: IEEE, 2010. p. 777–782.

MARTINS, M. et al. Open cell library in 15nm freepdk technology. In: **INT'L SYMPOSIUM ON PHYSICAL DESIGN. Proceedings**. [S.l.]: ACM, 2015. p. 171–178.

MARTINS, M. G. et al. Efficient method to compute minimum decision chains of boolean functions. In: **GREAT LAKES SYMPOSIUM ON VLSI. Proceedings**. [S.l.]: ACM, 2011. p. 419–422.

MARTINS, M. G. et al. Boolean factoring with multi-objective goals. In: **INT'L CONFERENCE ON COMPUTER DESIGN (ICCD). Proceedings**. [S.l.]: IEEE, 2010. p. 229–234.

MARTINS, M. G. et al. Spin diode network synthesis using functional composition. In: **SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI). Proceedings**. [S.l.]: IEEE, 2013. p. 1–6.

MARTINS, M. G.; RIBAS, R. P.; REIS, A. I. Functional composition: A new paradigm for performing logic synthesis. In: **INT'L SYMPOSIUM ON QUALITY ELECTRONIC DESIGN (ISQED). Proceedings**. [S.l.]: IEEE, 2012. p. 236–242.

MATOS, J. **Graph-Based Algorithms to Efficiently Map VLSI Circuits with Simple Cells**. Thesis (PhD) — Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil, 2017.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. Dag-aware aig rewriting a fresh look at combinational logic synthesis. In: **DESIGN AUTOMATION CONFERENCE (DAC). Proceedings**. San Francisco, CA, USA: ACM, 2006. p. 532–535. Available from Internet: <http://doi.acm.org/10.1145/1146909.1147048>.

MOCHO, R. et al. Asynchronous circuit design on reconfigurable devices. In: **SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN. Proceedings**. [S.l.]: ACM, 2006. p. 20–25.

MOREIRA, M. et al. Semi-custom ncl design with commercial eda frameworks: Is it possible? In: **INT'L SYMPOSIUM ON ASYNCHRONOUS CIRCUITS AND SYSTEMS (ASYNC). Proceedings**. Potsdam, Germany: [s.n.], 2014.

NEUTZLING, A. et al. Synthesis of threshold logic gates to nanoelectronics. In: **SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI). Proceedings**. [S.l.]: IEEE, 2013. p. 1–6.

NEUTZLING, A. et al. Threshold logic synthesis based on cut pruning. In: **INT'L CONFERENCE ON COMPUTER-AIDED DESIGN. Proceedings**. IEEE, 2015. p. 494–499. Available from Internet: <https://doi.org/10.1109/ICCAD.2015.7372610>.

NUNES, C. et al. Bti, hci and tddb aging impact in flip–flops. **Microelectronics Reliability**, Pergamon, v. 53, n. 9-11, p. 1355–1359, 2013.

OSORIO, M. C. et al. Enhanced 32-bit carry look-ahead adder using multiple output enable-disable cmos differential logic. In: **SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN. Proceedings**. [S.l.]: IEEE, 2004. p. 181–185.

PAPI. **Performance Application Programming Interface**. 2018. Available from Internet: <http://icl.cs.utk.edu/papi/overview/index.html>.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design: The Hardware / Software Interface**. 5. ed. [S.l.]: Morgan Kaufmann, 2014. 793 p. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 978-0124077263.

PERF. **Perf tool - Wiki**. 2018. Available from Internet: <https://perf.wiki.kernel.org>.

POLI, R. E. et al. Unified theory to build cell-level transistor networks from bdds [logic synthesis]. In: **SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI). Proceedings**. [S.l.]: IEEE, 2003. p. 199–204.

POSSANI, V. N. et al. Improving the methodology to build non-series-parallel transistor arrangements. In: **SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI). Proceedings**. Curitiba, Brazil: IEEE, 2013. p. 1–6.

POSSANI, V. N. et al. Graph-based transistor network generation method for supergate design. **Transactions on Very Large Scale Integration (VLSI) Systems**, IEEE, v. 24, n. 2, p. 692–705, 2016.

POSSANI, V. N. et al. Nsp kernel finder-a methodology to find and to build non-series-parallel transistor arrangements. In: **SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI). Proceedings**. Brasilia, Brazil: IEEE, 2012. p. 1–6.

PUGGELLI, A. et al. Are logic synthesis tools robust? In: **DESIGN AUTOMATION CONFERENCE (DAC). Proceedings**. New York, NY, USA: IEEE, 2011. p. 633–638.

REIS, A. et al. Extensive use of cmos complex gates with terminal suppressed bdds. **Journal of the Brazilian Computer Society**, v. 2, n. 2, 1995.

REIS, A. et al. Associating cmos transistors with bdd arcs for technology mapping. **Electronics Letters**, IET, v. 31, n. 14, p. 1118–1120, 1995.

REIS, A.; ROBERT, M.; REIS, R. Topological parameters for library free technology mapping. In: **BRAZILIAN SYMPOSIUM ON INTEGRATED CIRCUIT DESIGN. Proceedings**. Rio de Janeiro, Brazil: IEEE, 1998. p. 213–216.

REIS, A. I. Covering strategies for library free technology mapping. In: **SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN. Proceedings**. Natal, Brazil: IEEE, 1999. p. 180–183.

REIS, A. I. **Cell uniquification**. 2012. US Patent 8,214,787.

REIS, A. I.; ANDERSON, O. C. **Library sizing**. [S.l.]: Google Patents, 2011. US Patent 8,015,517.

REIS, A. I. et al. **Optimization of Integrated Circuit Design and Library**. 2010. EP Patent 2,257,900.

REIS, A. I. et al. **Optimization of integrated circuit design and library**. 2011. US Patent 8,024,695.

REIS, A. I. et al. Library free technology mapping. In: **VLSI: Integrated Systems on Silicon**. [S.l.]: Springer US, 1997. p. 303–314.

RIBAS, R. P. et al. Contributions to the evaluation of ensembles of combinational logic gates. **Microelectronics Journal**, Elsevier, v. 42, n. 2, p. 371–381, 2011.

ROSA, L. et al. Scheduling policy costs on a java microcontroller. In: **ON THE MOVE TO MEANINGFUL INTERNET SYSTEMS 2003: OTM 2003 WORKSHOPS. Proceedings**. [S.l.]: Springer Berlin Heidelberg, 2003. p. 520–533.

ROSA, L. S. da et al. Switch level optimization of digital cmos gate networks. In: **INT'L SYMPOSIUM ON QUALITY ELECTRONIC DESIGN (ISQED). Proceedings**. [S.l.]: IEEE, 2009. p. 324–329.

SCHMIDT, J.; FIER, P.; BALCáREK, J. On robustness of eda tools. In: **EUROMICRO CONFERENCE ON DIGITAL SYSTEM DESIGN (DSD). Proceedings**. Verona, Italy: IEEE, 2014. p. 427–434. Available from Internet: <http://doi.ieeecomputersociety.org/10.1109/DSD.2014.22>.

SCHNEIDER, F.; RIBAS, R.; REIS, A. Fast cmos logic style using minimum transistor stack for pull-up and pull-down networks. In: IEEE. **INT'L WORKSHOP ON LOGIC AND SYNTHESIS (IWLS). Proceedings**. Vail, USA, 2006. v. 15, p. 134–141.

SCHNEIDER, F. R. et al. Exact lower bound for the number of switches in series to implement a combinational logic cell. In: **INT'L CONFERENCE ON COMPUTER DESIGN: VLSI IN COMPUTERS AND PROCESSORS (ICCD). Proceedings**. [S.l.]: IEEE, 2005. p. 357–362.

SENTOVICH, E. M. A brief study of bdd package performance. In: **INT'L CONFERENCE FORMAL METHODS IN COMPUTER-AIDED DESIGN (FMCAD). Proceedings**. Palo Alto, CA, USA: Springer, 1996. p. 389–403. Available from Internet: <http://dx.doi.org/10.1007/BFb0031823>.

SILVA, D. D.; REIS, A. I.; RIBAS, R. P. Gate delay variability estimation method for parametric yield improvement in nanometer cmos technology. **Microelectronics Reliability**, Pergamon, v. 50, n. 9-11, p. 1223–1229, 2010.

SILVA, D. N. da; REIS, A. I.; RIBAS, R. P. Cmos logic gate performance variability related to transistor network arrangements. **Microelectronics Reliability**, Pergamon, v. 49, n. 9-11, p. 977–981, 2009.

TOGNI, J. et al. Automatic generation of digital cell libraries. In: **SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN. Proceedings**. [S.l.]: IEEE, 2002. p. 265–270.

VALGRIND. **Valgrind User Manual**. 2017. Available from Internet: <http://valgrind.org/docs/manual/valgrind_manual.pdf>.

VALGRIND. **Valgrind's Tool Suite**. 2018. Available from Internet: <http://valgrind.org/info/tools.html>.

WAGNER, F.; REIS, A.; RIBAS, R. Introdução aos circuitos digitais. In: **Fundamentos de circuitos digitais**. 1. ed. Porto Alegre, RS, Brazil: Sagra Luzzatto, 2006. chp. 1, p. 1–17. ISBN 85-241-0703-0.

WALLACE, D. E.; BLOOM, S. A. How can we build more reliable eda software? In: **INT'L WORKSHOP ON LOGIC AND SYNTHESIS (IWLS). Proceedings**. Berkeley, CA, USA: [s.n.], 2012. p. 8. Available from Internet: <http://bluepearlsoftware.com/files/Reliable_EDA_SW-IWLSfinal.pdf>.

WILTGEN, A. et al. Power consumption analysis in static cmos gates. In: **SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI). Proceedings**. [S.l.]: IEEE, 2013. p. 1–6.

YANG, B. et al. A performance study of bdd-based model checking. In: **INT'L CONFERENCE FORMAL METHODS IN COMPUTER-AIDED DESIGN (FMCAD). Proceedings**. Palo Alto, CA, USA: Springer, 1998. p. 255–289. Available from Internet: <http://dx.doi.org/10.1007/3-540-49519-3_18>.