



Politecnico
di Torino

ScuDo
Scuola di Dottorato - Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation

Doctoral Program in Software Engineering (36th cycle)

New Programming Paradigms for Optimization

Or Building Faster Programs for a Better Future

By

Andrea Calabrese

Supervisor(s):

Prof. Stefano Quer

Prof. Giovanni Squillero

Doctoral Examination Committee:

Prof. Alessandro Cimatti, ITC-Irts

Prof. Evelyne Lutton, Grenoble/INP Phelma

Prof. Cristina Manfredotti, INRAE Paris Saclay

Prof. Maurizio Rebaudengo, Politecnico di Torino

Prof. Mario Schölzel, University of Applied Sciences, Nordhausen

Politecnico di Torino

2024

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Andrea Calabrese

2024

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my parents, who have always been there for me.

*To my dear colleagues and friends, that supported me and beared me even at my
worst.*

To my friends (...and sorry if I wasn't present enough), listening to all my blabbering.

*In the end, to my supervisors. I wouldn't be here if they didn't believe in me, which
hopefully I did not disappoint... too much.*

*To Politecnico di Torino. We have hard times working together, but I share with it
some of the best friends I have.*

To ScuDo, which not even a stick to their heart can defeat.

To Sheila, the wonderful dog that freed me from my cage from time to time.

Abstract

Ph.D. est omnis divisum in partes tres: first of all, the testing branch. Then, the algorithm optimization and parallelization branch. Lastly, the Artificial Intelligence (AI) branch.

Testing. Chips are becoming larger and larger as new modules are added to improve performance and safety measures. In the testing domain, together with my colleagues at CAD group, we began developing tools for the evaluation of test programs. Those tools include an EVCD file analyzer, a toolchain built around it and a new metric for improving the speed of test routines development. The toolchain is able to speed up the ability to evaluate large test programs on an automotive System-on-Chip (SoC) from the SP58 family including more than 20 million gates, reducing the wall-clock time of execution from days to hours for larger files.

On the other hand, the new metric called *connectivity* is able to speed-up automatic test generation using a well-known evolutionary tool called μ GP. Moreover, it can also be used by programmers, providing a fast feedback with comparison to the standard fault coverage. However, we do not aim to replace the fault coverage metric, as it is the standard that provides complete information; thus, the connectivity, by computing partial information, is able to detect quick-to-fix errors in the execution of the program, providing instruction-level feedback to the test engineer.

Algorithms and Parallelization. Algorithms from NP class solve relevant problems today, and effort is put into improving their current performance. In particular, together with my colleagues and thesis students we focused on the Maximum Common Subgraph (MCS) problem, a well-known NP-hard problem that is used in molecule mining and even security. We improved the state of the art algorithm McSplit and its variants, among which the latest McSplitDAL, by applying the PageRank algorithm to classify vertices of the graphs. We were able to improve the original result up to 1.07 times on graphs of size up to 100 vertices. We also tested

different vertex classification algorithms on larger graphs of up to 7000 vertices, finding that, while PageRank is not always the winning metric, it provides the most stable improvements.

As Central Processing Units (CPUs) get more and more capable, and the multi-core paradigm became standard for large computations, also the many-core approach is becoming more important over the years: from Artificial Intelligence to computer graphics, Graphics Processing Units (GPUs) are ubiquitous in today's world, and research is being put into adapting or improving currently known algorithms into the many-core approach. In particular, together with my colleagues and thesis students we focused on the Graph Coloring (GC) problem, a well-known NP-complete mathematical problem. We were able to improve one of the state of the art algorithms on GPU, called Jones-Plassman-Luby, with a custom CUDA implementation, improving over the most famous implementation, Gunrock, up to 62 times, with a geometric mean of 3.16 times.

Artificial Intelligence. Together with Université Paris Saclay, we developed a system that is able to abstract rules using given knowledge of basic concepts. In particular, the objective of the system is building an inductive explanation of a game, detecting objects, categories and rules that the objects follow. Using a standard image detection library, OpenCV, we were able to feed videos involving 2 games, Arkanoid and Pong. By using a basic image recognition we were able to detect objects and their movements, detecting their velocities and positions. After this step, we were able to detect interactions between objects, creating a cause-effect relationship between interactions and change of status. In the end, we use a genetic algorithm provided by the Inspyred library to induce a set of categories and rules that apply to each category, involving interactions between them.

Contents

List of Figures	xi
List of Tables	xv
1 Introduction	1
1.1 Research activity	1
1.2 Technological introduction	3
1.2.1 Language: C++	3
1.2.2 Language: Rust	5
1.2.3 Language: Python	7
1.2.4 Concurrency and synchronization	7
1.2.5 GPU parallelism	11
1.2.6 Language: CUDA	13
2 Testing	14
2.1 Background	14
2.1.1 Testing	14
2.1.2 Value Change Dump files	16
2.1.3 Working with more VCDs	18
2.2 The VCD Analyzer	19
2.2.1 File reading	20

2.2.2	Pipelined parsing operations	21
2.2.3	Experimental results	24
2.2.4	Conclusions	31
2.3	The VCD Toolchain	31
2.3.1	The standard file	32
2.3.2	Layout-aware analysis	33
2.3.3	The Set Tool	38
2.3.4	The Hierarchical Analysis Tool	40
2.3.5	The Chip-Surface Stress Plotter	42
2.3.6	Experimental results	44
2.3.7	Experimental setup	45
2.3.8	The Logic Simulator	47
2.3.9	The VCD File Analyzer	48
2.3.10	Layout-Aware Elaboration Scripts	49
2.3.11	The Set Tool	55
2.3.12	The Hierarchical Analysis Tool	56
2.3.13	The Chip-Surface Stress Plotter	57
2.3.14	Wrapping up the flow	57
2.3.15	Conclusions	59
2.4	Connectivity: A new metric	61
2.4.1	The Proposed Methodology	61
2.4.2	The Basic Algorithm	65
2.4.3	Optimized Algorithm	69
2.4.4	Load/Store Instructions	71
2.4.5	Branch Instructions	71
2.4.6	Multiple Destination Instructions	75

2.4.7	Experimental Results	79
2.4.8	The Industrial Device under Analysis	79
2.4.9	Evaluation of SBST Programs	80
2.4.10	Evaluation of BI Programs	81
2.4.11	Evaluation of SLT programs	83
2.4.12	Conclusions and Future Works	83
2.5	Conclusions	84
3	Algorithms	85
3.1	Graphs	85
3.1.1	Notation	86
3.2	Graph Coloring on GPU	87
3.2.1	Introduction	87
3.2.2	Graph Coloring	89
3.2.3	Jones-Plassmann-Luby	90
3.2.4	Gebremedhin-Manne	92
3.2.5	Atos	97
3.2.6	Cohen-Castonguay	97
3.2.7	Gunrock	99
3.2.8	Our Coloring Procedure	101
3.2.9	Experimental Results	106
3.2.10	Performance analysis	117
3.2.11	Conclusions	119
3.3	McSplit+PR	120
3.3.1	Introduction	120
3.3.2	Background	122
3.3.3	McSplit	123

3.3.4	McSplit variants	125
3.3.5	Other Approaches	127
3.3.6	The PageRank Algorithm	128
3.3.7	The main idea behind our approach	129
3.3.8	McSplitX+PR	131
3.3.9	Experimental results	132
3.3.10	Conclusions and future works	136
3.4	Conclusions	139
4	A Core knowledge-based AI	140
4.1	Background	142
4.1.1	Core knowledge	142
4.1.2	Related works	143
4.2	Proposed approach	145
4.2.1	Objects identification	145
4.2.2	Evolutionary learner	148
4.3	Experimental evaluation	150
4.3.1	Implementation	151
4.3.2	Case study 1: Pong	151
4.3.3	Case study 2: Arkanoid	153
4.3.4	Behavioural considerations	155
4.4	Conclusions and future works	157
4.4.1	Towards the future works	157
5	Conclusions	158
5.1	A short wrap-up	158
5.2	Some final words	159

References

161

List of Figures

1.1	CPU-GPU workflow	11
2.1	VCD file structure	17
2.2	The proposed pipeline with all the stages. Colors represent how fast is a certain stage: green means faster, while red means slower.	22
2.3	The toolchain built around the VCD	32
2.4	The workflow for the layout-aware analysis	34
2.5	An example of the circuit as described in the VCD file.	35
2.6	An example of the clustering method (DBSCAN) executed on a generic layout	37
2.7	Rising and falling transition set cover: from file to set interacts	39
2.8	A high-level view of the tree data structure containing the coverage for each node	42
2.9	An example of stress heatmap over a generic SoC layout	43
2.10	Experimental setup	43
2.11	Coverage difference between exhaustive and deductive structural simulation for OpenRisc 1200.	47
2.12	Density-colored heatmap for the DUT	51
2.13	Runtime memory consumption of the heuristic method.	52
2.14	Evolution of BI metrics.	55

2.15	Visualization of the overall stress provided by the superimposition of all stress patterns.	58
2.16	Visualization of the overall stress provided by the superimposition of all stress patterns, plus the additional functional stress pattern targeting the identified unstressed module	59
2.17	Stress-colored heatmaps in terms of toggle coverage with detailed zoom on some regions.	60
2.18	The logic flow of our testing framework	62
2.19	A graphical representation of Example 2.4.1.	64
2.20	The CDFG of the code snippet of Table 2.23 in represented in Figure (a). Figure (b) illustrates the vertex-coloring process obtained following WAW edges, and Figure (c) the colors obtained at the end of the RAW visit.	67
2.21	An example including arithmetic operations and load/store instructions.	72
2.22	A graph example including a branch instruction.	74
2.23	Another branch instructions example.	76
2.24	A code snippet with instructions with multiple destinations: Our instruction-oriented analysis is modified to be destination-based. . .	77
2.25	The experimental setup including a development board with the SPC58 micro-controller and a hardware debugger.	80
2.26	Evolution of the population of ASM programs generated by the optimizer microGP.	82
3.1	An example of the Compressed Sparse Row (CSR) format: Graph (a) and corresponding CSR representation (b).	87
3.2	Application of the Jones-Plassmann-Luby algorithm on a small graph of 10 nodes.	92
3.3	Application of the Standard Gebremedhin-Manne algorithm on a 10 nodes graph	94

3.4	Application of the Gunrock implementation of the JPL algorithm on a 10 nodes graph	102
3.5	Application of our implementation of the JPL algorithm on a 10 nodes graph	106
3.6	Speedups of our implementations JPL_{\max} and $JPL_{\min-\max}$ against CuSparse, Gunrock, and Atos. The Gunrock procedure (in red color) is used as a reference and normalized to one.	113
3.7	Speedup of our $JPL_{\min-\max}$ approach (dealing with two independent sets for each iteration) over our JPL_{\max} methodology (dealing with a single independent set). The expected 2X factor is reached on average as overheads are negligible.	113
3.8	Elapsed time to complete each kernel launch within our $JPL_{\min-\max}$ strategy and the one delivered by Gunrock.	114
3.9	Percentage variations in the number of colors used by the different GPU-based methods with respect to GM_{s-imp} used as a reference and CPU-based.	117
3.10	The y-axis represents a percentage of the nodes, whereas the x-axis represents the execution progress.	118
3.11	Typical behavior of the effectiveness of the original implementation of McSplit. The size of the solution often increases rapidly in the first part of the process; then, the procedure is captured by local minima which slow down the convergence process and force the algorithm to visit enormous state spaces that do not improve the solution size. In orange, we can see the solution size at the end of the execution	133
3.12	The histogram plots the number of times each heuristic finds the MCS (i.e., the largest maximum common subgraph) on the 400 experiments. When a graph with the same size is returned by more than one method, each strategy is reported as a winner	134

3.13	A circular rolling average (with a window width of 50 consecutive tests) of the sizes of the solutions obtained by the McSplitX and McSplitX+PR algorithms on each instance. All values are normalized with respect to the results obtained by the original McSplit	136
3.14	The relative performance of the McSplitX and McSplitX+PR methods. For each row, we report the average improvement relative to the respective column. Darker blue colors highlight the size improvements	137
3.15	The dispersion of the points above the main diagonal shows that McSplitX+PR finds more extensive solutions in the vast majority of the cases	138
4.1	Proposed pipeline for finding general rules describing interactions between classes.	144
4.2	A video frame of Arkanoid shows the instant in which the ball hits the user's paddle and is about to bounce back.	146
4.3	A visual analysis of the correction of the ball velocity performed by our framework.	147
4.4	An image of the Pong game. The net (white stripes) divides the playing court in half; the two user-controlled paddles play against each other. The blue ball is represented in the middle of the court, and the scores are reported on the top of the screen.	152
4.5	The Pong puzzle: The classes and rules discovered by our strategy.	153
4.6	A video frame of an ongoing Arkanoid game, where some bricks have already been hit by the ball and disappeared from the image. .	154
4.7	The Arkanoid puzzle: The classes and rules discovered by our strategy.	154

List of Tables

2.1	Single point full timing analysis. Running times for the different pipeline stages, with an increasing number of threads, for the smaller VCD file, i.e., the one of 38 GBytes. All times are reported in seconds. The symbol – means that the data is meaningless in that experiment.	25
2.2	Single point full timing analysis. Running times for the different pipeline stages, with 32 threads (our most efficient configuration) for VCD files with increasing size (from 10 to 243 GBytes). All times are reported in seconds or hours (when explicitly stated).	25
2.3	Single point full timing analysis. In-depth analysis of our approach: Waiting times for the two most critical phases (the Read and Parse stages), pipeline efficiency, and speedups for the Parse phase and the entire process. All times are reported in seconds or hours (when explicitly stated).	26
2.4	Single point stress timing analysis. Running times for the different pipeline stages of our chain, with an increasing number of threads for the smaller VCD file, i.e., the one of 10 GBytes. All times are reported in seconds. The symbol – means that the data is meaningless in that experiment.	27
2.5	Single point stress analysis. Running times for the different pipeline stages, with 32 parsing threads (our most balanced configuration) for VCD files with increasing size (from 10 to 243 GBytes). All times are reported in seconds or in hours (when explicitly stated).	27

2.6	Single point stress analysis. In-depth analysis of our approach: Waiting times for all threads of the two most critical phases (read and parse stages), pipeline efficiency P (Equation 2.1), and speed-ups for the Parse phase and the entire process. All times are reported in seconds.	28
2.7	Multiple Point Stress Analysis. Running times for the different pipeline stages, with different thread configurations, for the smaller VCD file, i.e., the one of 10 GBytes. All times are reported in seconds. The symbol – means that the data is meaningless in that experiment.	29
2.8	Multiple Point Stress Analysis. Running times for the different pipeline stages. with the 16/64 threads configuration (the most efficient one in our experiments) for VCD files with increasing size (from 10 to 243 GBytes). All times are reported in seconds or hours (when explicitly stated).	29
2.9	Multiple Point Stress Analysis. In-depth analysis of our approach: Waiting times for all threads of the two most critical phases (read and parse stages), pipeline efficiency, and speed-ups for the Parse phase and the entire process. All times are reported in seconds or hours (when explicitly stated).	30
2.10	A simplified view of the coverage file, highlighting the information looked by the hierarchical analysis tool	41
2.11	The logic simulation phase: CPU times and size of the VCD files generated. The symbol “*” means the time is estimated. “N/A” means that the value is Not Available.	45
2.12	Profiled execution for the VCD Analysis for the single-point stress metric.	48
2.13	Profiled execution for the VCD Analysis for the multi-point stress metric.	49
2.14	Profiled execution of Virtual node elimination script.	50
2.15	Comparing the execution time of the exhaustive and the heuristic approach.	52

2.16	Confusion matrix showing the accuracy of the heuristic method compared to the exhaustive.	53
2.17	Single point stress metric coverage.	54
2.18	Multiple point stress metric coverage.	54
2.19	Profiled execution of the Set Covering tool.	56
2.20	Profiled execution of Hierarchical analysis tool.	56
2.21	Profiled execution of Chip-surface stress plotter.	58
2.22	An example of instruction sequence with the operation performed.	63
2.23	Instruction sequence, source, and destination operands.	66
2.24	Instruction sequence, source and destination operands.	71
2.25	Instruction sequence, source and destination operands.	73
2.26	Instruction sequence, source and destination operands.	75
2.27	Instruction sequence with multiple destinations.	76
2.28	Test programs evaluation. All branches without an alternative are marked as black in our analysis. NA means the data is not available.	79
3.1	The main characteristics of the benchmark graphs used during our experimental analysis. The graphs are numbered from 1 to 28 to find an easy correspondence in the following plots. Column <i>Type</i> indicates the main characteristics of each graph: Real (r) or generated (g), undirected (u) or directed (d), following a power-law degree distribution (p) or not (-). Notice that power-law graphs, which have distinct characteristics and on which the different approaches behave differently, have been inserted in the second part of the table.	107

3.2	Average coloring time for each one of our implementations. Columns' headers have the meaning described in the itemization included in the main text. On the CPU-side, we indicate with GM_{s-imp} and GM_{a-std} our implementations of the GM synchronous and asynchronous algorithm, and with $JPL_{min-max}$ the JPL procedure. On the GPU side, we report the results of cuSparse, Gunrock, and Atos. The last two columns include our implementations on GPU. Once more, the graphs after the horizontal line (used as a separator) follow a power-law degree distribution.	108
3.3	Detailed comparison of our JPL min-max approach against Gunrock. Notice that Gunrock pre-processing times are roughly equivalent to the sum of our times, including the vector randomization, the GPU allocation, and the CPU-to-GPU transfer time. The post-processing phase includes the GPU-to-CPU transfer time.	111
3.4	The average number of colors for each one of our implementations. On the CPU side, we present our implementations of the GM synchronous and asynchronous algorithm (GM_{s-imp} and GM_{a-std} , respectively), and the JPL procedure ($JPL_{min-max}$). On the GPU side, we report the results of cuSparse, i.e., the csrColor Cohen and Castonguay implementation, Gunrock, i.e., the Gunrock's algorithm, and Atos from Chen et al. The last two columns include our implementations on GPU.	116
3.5	Percentage of instances improved by the PR methods (columns) over the original methods (rows), without breaking ties	134
4.1	The evolution of Pong and Arkanoid games respectively: for each seed, we show the fitness of the best individual in the first generation and the fitness of the best individual in the last generation	156

Chapter 1

Introduction

Central Processing Units (CPUs) are becoming more and more capable every generation. Vectorization units, multi-core CPUs and many-core Graphic Processing Units (GPUs) are considered standard elements of today's computers. Moreover, GPUs are becoming more and more capable, and much progress has been made in the General-Purpose Computing on Graphics Processing Units (GPGPU), to the point that GPUs can run arbitrary code. Due to this fact, parallel computing is becoming much more important, both on the multi-core approach and on the many-core approach.

My Ph.D. work develops on 3 different branches of computer sciences: first, the *hardware testing* branch; then, the *algorithmic optimization and parallelization* branch; in the end, there is a small incursion in the *Artificial Intelligence* branch. As we will see, however, the algorithm parallelization pervades all of the branches, standing as the protagonist of this thesis.

1.1 Research activity

In this section, I will focus on a brief description of the various chapters: the main subject, the goal of each one and a short description of the results.

Chapter 2 focuses on the hardware testing branch. In the hardware testing branch of my work, I mainly focus on the evaluation of test programs. In particular, my work involves the automated evaluation of test programs according to standard metrics already known in the hardware testing field (such as the *toggle activity* of

signals) using a standard file format (VCD) and the creation of a new metric, called *connectivity*. The main goal of this work is speeding-up the development of tests on hardware platforms.

In the VCD work, the main goal was analyzing metrics such as the toggle activity in as few time as possible, as VCD files grow very quickly in size. As soon as the VCD analysis was ready, I focused on creating a testing environment based on VCD files, creating a toolchain able to filter, merge, visualize and classify the faults that may be present in the circuit.

In the connectivity work, however, the objective was creating metric that was quick to compute, but that could give some information about the fault coverage of the tests on a circuit, since the fault coverage itself requires a significative amount of time to be computed. Connectivity, instead, is very quick to compute and provides information on the propagation of possible errors during the execution of a test program. Moreover, the connectivity metric can also be enhanced by the analysis of bit-level propagation of the data, effectively making it a closer measure to the actual fault coverage.

Chapter 3 focuses on the algorithmic work, namely the improvement of the state-of-the-art algorithm for the Maximum Common Subgraph and the porting and comparison of different Graph Coloring algorithms on GPU.

The state-of-the-art algorithm for computing the Maximum Common Subgraph is called McSplit. Through the usage of the PageRank algorithm before the execution of McSplit to classify the vertices of the graph, I was able to improve the solution of the algorithm on a large set of graphs, either by reaching a good solution in less time or by reaching a solution including more vertices of both graphs.

For the other subject, the objective is to improve the execution time of the Graph Coloring task through the usage of a GPU. First, we ported the JPL algorithm as-is to the GPU; then, we improved it by adding another step to each coloring cycle, which made the algorithm perform much better than earlier. In the end, we compare against different implementations of JPL both on GPU and CPU and against the algorithm GM, both the CPU version and the GPU porting that uses the Atos framework. In this case, we were able to improve on all the other variants of JPL, while GM is the leading algorithm on larger graphs.

Chapter 4, instead, focuses on the creation of an AI approach based on the theory of the Core Knowledge. In particular, in this approach we define some concepts that are known in advance and try to extrapolate some rules based on those concepts. In our case studies, we chose 2 classic videogames to analyze: an Arkanoid clone and a Pong clone. These games make use of some physical concepts, such as velocity, bounces and collisions, and some more abstract concepts, such as the appearance and disappearance of objects. The approach uses a genetic algorithm to detect the rules through the analysis of a sequence of events, available by analyzing a gameplay video of those games. Most of the analysis difficulties, however, reside in the translation of the video to the events, as the video presents aliasing problems due to the pixel sampling, which is a discrete interface, while the effective values of the games are in the continuous domain. Despite this, the approach is able to infer some reasonable rules, recognizing, for instance, the disappearance of a block once the ball bounces on it.

1.2 Technological introduction

In this section, I will provide an introduction to the technologies used during my Ph.D. In particular, I will focus on programming languages and paradigms that we need for the following chapters, not considering domain-specific knowledge.

In Section 1.2.1 we will talk about C and C++ and their characteristics. In Section 1.2.2 we will talk about the Rust programming languages and its features. In Section 1.2.3 we will briefly describe the Python programming language. In Section 1.2.4 we will describe concurrency and synchronization primitives both on CPU and on GPU. In Section 1.2.6, we will talk about CUDA, a language used for writing programs that run both on CPU and GPU.

1.2.1 Language: C++

C++ [1] is a programming language born as a superset of C with the intent of providing compatibility with C and introducing the notion of *classes*, providing an object-oriented approach. As C is a language well-known for its flexibility and closeness to machine code, it provides a very limited standard library compared to more modern languages. In particular, C++ provides elements such as fully-typed

dynamic arrays, linked lists, hash maps, sets and much more. C++ also allows for Resource Acquisition Is Initialization (RAII) paradigm, which is strongly used from C++11 onwards; this paradigm exploits the scope of the variables to manage resources, such as *smart pointers* or *locks*.

Moreover, as C revisions added few utilities over the years, C++ standards brought many changes over 2 decades. The most noticeable C++ versions are C++03, C++11, C++17 and C++20. While each version brought noticeable improvements, the leap from C++03 to C++11 is regarded as the largest one in the community, as it added in-language support for multithreading, multitasking and smart pointers. C++17 introduced a common interface for the filesystem and the `constexpr` keyword, and C++20 added the concept of modules and the `constexpr` keyword. We are mostly interested in the changes performed by C++11, and a limited subset of C++17, mostly including the *constexpr* construct.

In particular, for C++11, we are interested in:

- `std::array`: the C++ array is a statically sized array that acts as a C array with the difference of having the information of the size included and accessible through the container interface.
- `std::vector`: the C++ standard vector is a strongly typed dynamic array with automatic resizing that provides a good trade-off between memory utilization and performance. Moreover, it calls the constructor of the classes inserted when resizing.
- `std::unordered_map`: it is an implementation of a hashmap, using hashable elements as keys and elements of any type as value.
- `std::thread`: the C++ standard thread is an operating system independent implementation of the thread library. It is a convenient abstraction over system calls, that uses the `pthread` library on Posix systems and the Windows thread library on Windows. More on threads and synchronization in Section 1.2.4.
- `std::mutex` and `std::lock`: the former is a mutual-exclusion (as the name implies: mut-ex) synchronization primitive among threads. The latter is a RAII container over synchronization primitives, in which locks depend on the existence of the variable; as it locks a mutex on a thread, the lock is automatically removed when the variable goes out of scope.

For C++17 we are mainly interested in the `constexpr` keyword. This keyword has two meanings, depending on its usage: if used on a variable, it declares the variable as a constant and it forces the compiler to replace it with an immediate number at compile time. Both C and C++ have the concept of a constant, but the compile time substitution using the `const` keyword is merely a suggestion and not a directive. The second usage of `constexpr` is in front of functions: in this case it suggests the compiler to execute the function at compile time if possible. However, if the function is not executed at compile time, no error is produced and the compiler produces a non-`constexpr` function instead. C++20, with the introduction of `constexpr`, explicitly forces the compiler to execute the function at compile time if possible, otherwise producing an error.

1.2.2 Language: Rust

Rust [2] is a modern programming language built around memory safety and modern programming paradigms that are considered good practices, such as RAII. It differs by C-like programming languages in some important ways, as this safety comes with a shift in programming paradigm. It is divided in two modules: *safe* and *unsafe*; we will discuss the safe part, as the unsafe one is mostly used to interface with C and C++ libraries or for lower level programming. Some of the main characteristics of Rust are:

- Memory ownership model: in opposition to C and C++ memory model, which does not require the concept of ownership, as memory is considered as a large contiguous array, Rust enforces the ownership concept: every data owns/is owned by other structures, implying a single instance of that data exists. This does not imply that data can not be copied among different structures, but it has to be done explicitly.
- Move as default: due to the ownership memory model, the meaning of the assignment operator among variables = is a *move* operation. Once data is moved, the previous variable holding the data becomes invalid; this is true also for parameter passing among functions. Instead, copies need to be explicit; this ensures that the programmer understands that copying comes at a price, and avoids useless copies as much as possible

- Immutability by default: in opposition to most programming languages, variables are immutable by default. To mutate them, they have to be declared with the `mut` keyword.
- Zero-cost abstractions: Abstractions through interfaces in most object-oriented languages imply the concept of inheritance. Inheritance, however, comes at a cost: in particular, in C++ a *virtual methods table* (v-table) is created at compile time, and at runtime, based on the type of the data, the right function is executed. However, the cost of the v-table is a dereferentiation of function pointers together with multiple jumps through the code (one for the v-table, another one for entering the desired function). Rust enforces interfaces through the use of *Traits*, which define methods of the interface belonging to the data structure. Traits are not treated as dynamically dispatched functions, and their meaning is only programmer-sided: the compiler only sees the implementations inside and compiles them as normal functions.
- Smart pointers: as ownership is the default memory model, smart pointers are the most common way to access data on the heap. There is overlapping among C++ `std::unique_ptr` and Rust's `Box`, and among C++ `std::shared_ptr` and Rust's `Rc` (Reference Counting) pointer. However, for thread safety, Rust provides a new pointer: `Arc` (Atomic Reference Counting), which allows for data to be shared among threads. Since all smart pointers use the RAII paradigm as default, null or invalid pointers do not exist in *safe* Rust. To deal with optional values, Rust provides a wrapper called `Option`, that needs to be explicitly checked before being dereferenced.
- Explicit result handling: as Rust does not have an exception system, it provides the wrapper `Result` for code that may produce recoverable errors; this allows for the error management to be both explicit and lightweight, often not even requiring heap allocations.
- Fearless concurrency and thread safety: through the use of the traits `Send` and `Sync`, Rust allows for objects to be shared among threads. However, as a single instance of any object needs to exist, shared objects are often wrapped in `Arc` if data needs to be accessed by multiple threads. However, by exploiting both the Move as default and the Immutability as default paradigms, data inside Arcs need to be immutable. However, synchronization data structures such

as `Mutex` and even entire paradigms such as `RwLock` (Readers and Writers Lock) can be used as wrappers for the data inside the Arc, providing both mutability and thread-safety against data races.

- Standard build system: the program *Cargo* handles the projects in Rust, and is responsible of managing all the external dependencies, downloading them from the *crates.io* repository if needed. Its build system is also flexible and allows for user-defined modifications to compile libraries that might need an internal toolchain.

All of the points mentioned above provide what is called *memory safety*. Moreover, the move as default model is enforced by a part of the compiler called *borrow checker* (as moving is often called "borrowing" in Rust slang). The borrow checker analyzes loops and functions both for concurrent and sequential code to search for use of invalid values; this is done at compile time, in contrast with C++, in which a `std::move` function exists, but invalid references are allowed and must be checked at runtime.

1.2.3 Language: Python

Python [3] is a scripting language that became famous for its ease of usage. It is one of the most used programming languages in the world according to PYPL [4], which uses data from Google Trends to perform each language popularity ranking. However, this comes at the cost of performance, being much slower than lower-level programming languages such as C or C++. However, due to its widespread usage and its many libraries, it is mostly used for Machine Learning tasks and web development. Its type system is often called "duck-typing", as "If it looks like a duck and quacks like a duck, it must be a duck." [5] This provides simplicity at the cost of safety and performance.

1.2.4 Concurrency and synchronization

In this section, I will discuss about concurrency on modern CPUs. I will discuss about how its different strategies, each with its pros and cons, synchronization mechanisms and some common paradigms that can be implemented.

Concurrency

Concurrency is the ability to execute programs in an out-of-order fashion. Historically, on single-core CPUs, concurrency was already performed by the various operating systems. One of the main reasons for concurrency in single-core CPUs was providing a way to perform work instead of waiting for resources to be available (for instance: reading and writing in memory is an expensive operation, as it involves synchronizing data between physically distant hardware pieces). However, concurrency can also be implemented as parallelism [6], although it is not a necessary statement; however, in category theory, parallelism is a sub-category of concurrency.

In particular, parallelism is the ability to distribute execution units among different processing units (for instance, CPU cores). Today, parallel computing is becoming more and more popular due to the increased parallel capabilities of CPUs; it is not unusual for today's CPUs to have 4 or more CPU cores even in mobile devices, sometimes even divided between high-performance and high-efficiency cores. In servers, we can have even thousands of CPU cores, and multiple CPUs, on the same machine. Concurrent and parallel computing can take various forms:

- **Multi-processing:** as each process is possibly assigned to a different CPU, and process execution is interleaved, it is possible to exploit having multiple processes in execution cooperating towards an objective. The drawbacks of this approach is that processes have a high overhead to be created and do not share memory: inter-process communication needs to be set up. Moreover, as processes can be cloned, terminating the parent process does not terminate the child processes, which have to be terminated explicitly.
- **Multi-threading:** threads are lightweight concurrent routines that are executed within a single process. Thread creation provides much less overhead than process creation. Moreover, they can share memory and resources with each others. This raises new problems with data access, such as data races, which we will discuss.
- **Task-based concurrency:** a task is a lightweight routine running inside a thread. Tasks are easily created and later assigned to threads. Tasks are often called *asynchronous* functions, and the implementation of task-based concurrency depends on the programming language and its libraries. For example, both

C++ and Rust use a single thread to handle asynchronous tasks. However, some libraries have been built for both languages to exploit the multi-threading capabilities of the CPUs, providing concurrent and parallel tasks.

As parallel computing is often associated with multiprocessing or multithreading, it also comes in form of Single Instruction on Multiple Data (SIMD) modules for the CPU, such as the AVX on Intel, or Luna on ARM CPUs. Those modules are often exploited by compilers if the code is mostly simple. These modules consists in special contiguous registers and instructions that act on them. However, there is a trade-off between portability and performance, as either assembly language or intrinsic C functions are needed to use those modules explicitly. To avoid this trade-off, we decided to design our data structures to help the compiler understanding what can be performed using SIMD operations.

The performance gain, however, is not linear and is subject to Amdahl's law [7]. This is due to the need of synchronization (more on this subject in Section 1.2.4) and overheads due to the creation of the concurrency primitive (processes, threads and tasks). However, there is a class of problems that performs close to the limit of this law, called *embarrassingly parallel problems* [8], in which each concurrency primitive is independent and work can be performed without need from synchronization, with the exception of a barrier at the end of the computation.

Synchronization

Synchronization is a task that coordinates the execution of different concurrency primitives. Synchronization also has many primitives; here we will discuss about some of them:

- Semaphores [9]: they are the most general-purpose primitive. Theorized by Dijkstra, semaphores allow for resource protection and avoidance of data races. A semaphore has an internal counter that can be increased or decreased. When it reaches 0, however, it stalls the current concurrency primitive (internally asking the scheduler to switch to another process). Semaphores are usually implemented for thread synchronization on Linux systems, although the standard does not explicitly mention this.

- **Mutexes:** the name stands for *mutual exclusion*, and can be implemented with semaphores with a maximum count of 1. Mutexes can be either locked or unlocked. If many synchronization primitives try to lock the mutex, only one can lock it, while the others will wait until it gets unlocked.
- **Spinlocks:** a spinlock, also called busy waiting, is a synchronization primitive that keeps performing a check on a variable until it reaches a certain value. This allows for synchronization in its most basic form, but its usage is generally regarded as a bad habit. However, they provide the advantage of not needing to invoke the scheduler, which allows for a very fast response time.
- **Atomic variables:** an atomic variable is a variable on which operations are always performed sequentially. In practice, they are implemented inside CPUs in special registers, and allow for the existence of the other synchronization mechanisms.
- **Condition variables:** often used together with a mutex, they provide a mutex lock until a condition is satisfied.

Thanks to the synchronization mechanisms, parallel paradigms rose to solve common problems. Among those, we use:

- **Producer-Consumer:** this paradigm is used to pass work among threads, distributing it and balancing each task. For instance, we can have a thread that reads from a file partial data producing a string, and a second thread that consumes that data creating a complex structure.
- **Readers-Writers:** when protecting a resource, we may want to lock it only from destructive operation, such as a write, while reading data in parallel is perfectly fine. This paradigm provides a lock of a resource only in presence of writers. Variants of this paradigm exist, giving priority to readers or to writers; however, the most common implementation prioritizes reading operations.
- **Pipeline:** a pipeline approach consists of more producer-consumers in sequence. This can balance consumers that need to deal with expensive operations.

1.2.5 GPU parallelism

GPUs provide massive parallelism similarly to the SIMD paradigm. In this case, we talk about Single Instruction on Multiple Threads (SIMT), as GPUs provide thread scheduling out of the box. In particular, GPUs are very good at performing the same work on a very large amount of data due to their design. Historically, GPUs were designed to perform few operations, such as matrix multiplications. This design was reflected by earlier graphics APIs, such as older versions of OpenGL and DirectX. Most notably, with the advent of programmable shaders, researchers such as Krüger et al [10] were able to perform general purpose computing on a GPU, however resorting to a reformulation of the problem using graphics primitives. This represented the first stone put towards high-performance computing, leading to the creation of CUDA from Nvidia and OpenCL from the Khronos Group and Apple [11]. However, CUDA only works on Nvidia GPUs, while OpenCL (and, more recently, Vulkan [12, 13]), work across many GPU vendors. In our work, we used CUDA, as its language is very similar to C++ and even allows for seamless code interoperability through header files. However, the concepts and limitations on the working principles of GPUs remain valid for all programming libraries.

Figure 1.1 shows the typical workflow between CPU and GPU: first, the CPU provides the data and the program to be executed on the GPU; then the GPU executes the program; in the end, the CPU retrieves the results from the GPU.

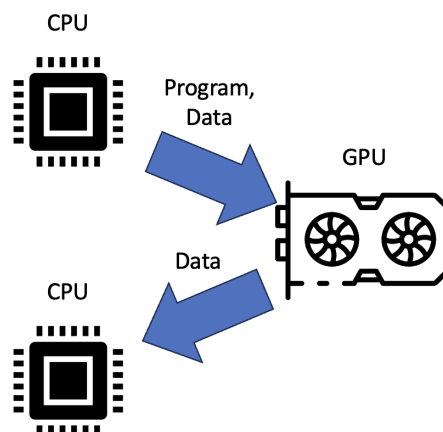


Fig. 1.1 An example of the typical workflow using CPU and GPU

GPUs provide many advantages in term of computation. In particular:

- they provide massive parallelism, as mentioned above. Less recent Nvidia GPUs of the GTX series provided 1024 cores. However, with the more recent RTX, depending on the model we can get over 8192 cores working simultaneously.
- Operations on small amounts of data are performed using a SIMD approach. Thus, operations such as small vector addition are performed simultaneously, providing further optimization possibilities.

However, GPU computing comes with many limitations [14]; in particular:

- Operations on a small amount of data tend to perform very poorly; this is because of the hardware design of the GPUs, trading off raw single-thread performance for multi-core performance.
- Copying data between CPU and GPU on non-unified memory architectures is expensive; mechanisms cope with this limitation, such as asynchronous data copy and execution, are present in all the most common programming libraries.
- Due to its hardware architecture, branching on GPU is a potentially long operation, as branching paths are executed sequentially, with the exception of recent Nvidia GPUs that interleave the execution of different branches, trying to save time from stalls.
- Moreover, GPU cores lack a branch prediction unit, so even the usage of loops has to be evaluated when writing an algorithm to the GPU.
- Due to its hardware architecture, threads are spawned in multiple of 32 on Nvidia or 64 on AMD graphics cards; in general, this is called a work group, but each vendor has its own term (Nvidia uses the term *warp*, AMD uses the term *wavefront*). This can lead to spawning threads that perform no work, however providing more stress to the scheduler.
- Reading and writing operations on memory are usually slow compared to computations. Thus, accessing memory should be avoided as much as possible.

This shows that porting parallel algorithms from CPU to GPU is not always trivial, and it is difficult to analyze the advantages of one approach over the other.

Synchronization primitives within are also mostly technology dependent. For instance, CUDA 12 provides atomic operations and graph-based execution, in which it is possible to declare an execution graph of various kernels. However, synchronization mechanism keep changing even within CUDA as technology is evolving, thus leading to very version-specific methodologies even within the same framework.

1.2.6 Language: CUDA

CUDA [11] is a language similar to C++ developed by Nvidia, which allows, with a syntax similar to C, to perform general-purpose computing operations on a Nvidia GPU. Its parallel functions are called *kernels*. Calling a kernel from the CPU provides explicit control over the number of threads spawned on the GPU. However, some operations, such as dynamic allocation of GPU memory, are allowed only on CPU-side. As seen in Figure 1.1, CPU and GPU work uses a fork/join paradigm. In CUDA, this is performed by default implicitly with a kernel call, for which the syntax is similar to function calls.

However, CUDA also provides functions that allow for asynchronous operations, enabling both streaming operations, such as copying and computing performed as soon as data becomes available, and parallel operations among CPU and GPU. Those operations can be performed through the usage of the *Async* suffix in the function call for memory allocations and copy, also providing a structure called *CUDA Stream*. A CUDA Stream effectively tracks the execution of the GPU workflow for which it has been used. CUDA Streams provide the capability of performing simultaneously work on both GPU and CPU, at the cost of an explicit waiting operation on the stream itself. Moreover, they provide the GPU enough information to perform even partial work as soon as the data is ready, enabling increased concurrency capabilities inside the parallel execution. Theoretically, there is no upper bound for the number of operations in a stream or for the number of streams that the programmer is able to create.

Chapter 2

Testing

This chapter describes the publications [15–20]. In particular, I will show a tool for the analysis of large simulation dump files, a toolchain built around it and a new metric for analyzing test programs. Those works share a common background, which I will describe in Section 2.1. These works have been developed together with my colleagues Francesco Angione and Lorenzo Cardone, and with professor Paolo Bernardi. Moreover, in the evolution of the VCD tool, a special mention should go to my colleagues Gabriele Filippini and Giusy Iaria, who helped me finding bugs and improving the tool over the time.

2.1 Background

2.1.1 Testing

Systems-on-Chip (SoCs) are becoming more and more large as technology advances, including a huge number of gates. Together with those chips, we witness an increase of failure mode strategies. Some of the issues that arise with this increase in scale are the simulation time and analyses. In particular, fault simulations tend to be very expensive, leading to entire weeks of computation. This is because modern SoCs include computational units, co-processors, memories and many more modules. Modules are verified and validated at each phase of development, from design to in-field testing. We focus on the latter. However, even if it is only one phase, optimizing

it may lead to a significant decrease of analysis time, as the computational effort is significant [21].

In the automotive scenario, devices should also respect the safety requirements dictated by the standard ISO-26262 [22], every defect potentially causing a harmful failure should be avoided. Therefore, for automotive SoCs, the standard manufacturing test flow, usually encompassing only wafer and package tests, is enhanced with a Burn-In stress step, followed by a final package test, and eventually a System-Level-Test phase [23–25]. In this article we essentially concentrate on the Burn-In phase. Still, we show that our analysis flow can also provide useful feedback about final test procedures and SLT.

Burn-In (BI) is a stress phase designed to remove the infant mortality of SoCs [26] manufactured with some weakness, such as thinner metal oxide or metallization. The BI phase provides both external and internal stress to a device. External overexertion (such as thermal stress) is generated by a climatic chamber or at the socket level. Its main objective is to age the device material [27]. Internal stress (such as electrical stress) is produced by scan-based approaches [28], Built-In Self-Test (BIST) modules [29], or functional test programs [30, 31]. Its main target is to force device gates to produce high internal activity and possibly exacerbate the insurgence of latent defects not screened by the wafer and packaged test procedures, which could be later captured by the successive test steps.

Nowadays, grading the quality of BI procedures is getting extremely important [23, 32]. Unfortunately, grading activities are affected by the size and complexity of modern SoCs. Moreover, modern devices present new defect types [33, 34] requiring enhanced coverage metrics not fully supported by current Electronic Design Automation (EDA) tools. For example, the most frequently used metric, i.e., the so-called toggle coverage [35], represents the number of gates that make at least one transition. Nonetheless, there is an increasing interest in also evaluating the average number of times and how uniformly a signal is stressed [36]. To perform these advanced tasks, it is necessary to simulate the stress patterns and post-process the resulting simulation dump [16]; this approach may become prohibitively expensive for large devices. Furthermore, the SoC layouts should be considered as gate density may vary from area to area, affecting the accuracy of the stress measures [37] [35].

To perform different tests on SoCs, manufactures rely on Automatic Test Pattern Generators (ATPGs) to generate test patterns and perform fault simulations using

those patterns on appropriate fault simulators [38–40]. ATPGs rely on few standard stress metrics and can efficiently perform a first pass of tests, with a knowledge of the chip layout. However, it is possible to perform tests even with pre-defined patterns using fault simulators, that may be generated with a deeper knowledge of the chip by the manufacturer. Moreover, on-line testing methodologies do not use ATPGs, as the cost would be excessive for a real-time system, but rely on testing routines; this is called *functional testing*, and it often requires a deep knowledge of the chip architecture. Nonetheless, functional testing is a vital part of the testing procedure and may be performed both for chip validation and for on-line testing.

2.1.2 Value Change Dump files

A Value Change Dump (VCD) [41] file is a trace of all the changes on the gates and buses of a SoC during the test. The VCD file is logically divided in 3 main sections:

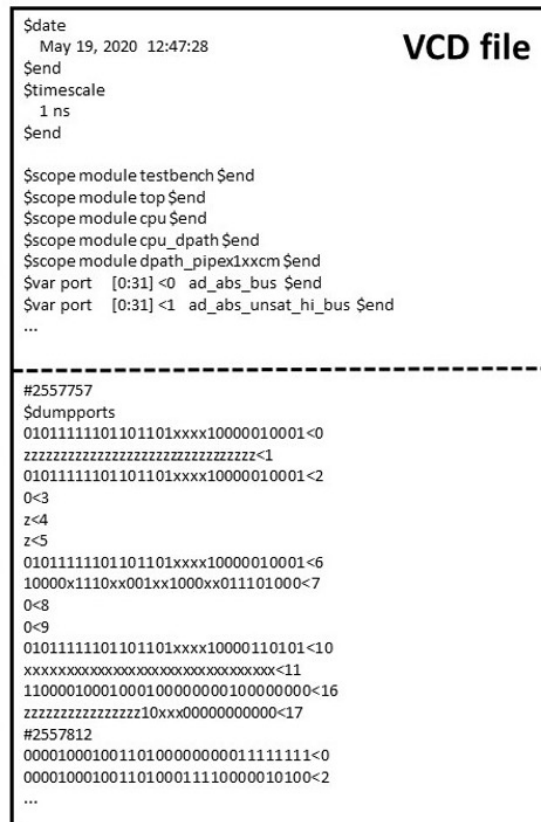
- Signal and hierarchical declaration of the SoC circuit, which introduces the signals that the circuit will pass and assigns them an internal reference number. The number of the signals is usually
- An initialization of the signals at time t_0 . This section is mandatory for all signals.
- The changes of each signal divided by time frame. However, to save space, the VCD standard provides, for each time frame, only the signals that change.

Moreover, the following year the Extended VCD (EVCD) [42] format was proposed. The main difference between the two formats is the ability of the EVCD format to express the strength of the signals, also providing information about the input and output states using different symbols.

Figure 2.1 shows an example of the file structure generated by the tool SimVision (also known as NCSim) by Cadence [43], which mixes VCD and EVCD syntaxes. In Section 2.1.3 we will highlight the difference between various VCD formats.

Every VCD file is divided in 3 parts: first, the declaration and initialization of the signals. The *\$scope module* directive declares the beginning of a module, terminated by an *\$upscope*. In this case, each signal is declared using the *\$var port* directive, and it contains information about its size (thus, we can deduce if it is a bus or a gate),

its internal reference number and the name of the signal within the hierarchy. The second part and the third one share the same format, and are shown after the dotted line on the image. Each timestamp is written on a line beginning with the character '#'. Then, each line shows the new value of the signal before the character '<', and the internal identifier of the signal after the separator.



The image shows a screenshot of a VCD file structure. The top section contains metadata: \$date (May 19, 2020 12:47:28), \$send, \$timescale (1 ns), and several \$scope lines for modules like testbench, top, cpu, and dpath. It also lists \$var port for signals like ad_abs_bus and ad_abs_unsat_hi_bus. A dashed line separates this from the signal data section, which starts with a timestamp #2557757 and shows \$dumpports for the same signals, with values like 0101111101101101xxx10000010001<0 and 0101111101101101xxx10000010001<2. The data continues with more timestamps and values, ending with #2557812 and values like 0000100010011010000000011111111<0.

Fig. 2.1 The file structure of the VCD file.

In the VCD standard, a signal can be a gate or a bus. The main difference is that gates can hold only 1 value, while a bus can hold an arbitrary number of values. Moreover, each value represents 1 bit, representing the state of the signal from the simulation. Values can be:

- logical 1.
- logical 0.
- x: an unknown value among logical 1 and logical 0; this value means that the gate is an undefined state.

- z: high impedance, usually meaning that that gate is not supplied.

2.1.3 Working with more VCDs

As VCD is a well-defined standard, in practical cases the standard must not be taken from granted when working with different EDA tools. Open source and free tools, such as GTKWave [44], are aware of this issue and choose to support a limited number of formats. In particular:

- According to the VCD standard, signal IDs include symbols and alphanumeric characters. This usually leads to more compact IDs for the signals.
- Gate changes do not provide a separator between the new value of the signal and its ID, and are written in the format *<change><ID>*.
- Bus changes start with a 'b' character and list all the values in the bus; then, the ID is separated by a whitespace from the changes to avoid ambiguity, resulting in the format *b<changes> <ID>*.

Instead, in the EVCD standard:

- Signal declarations use '<' as separator between the gate/bus size and the signal ID.
- The signal ID is represented by continuously increasing integers.
- Each change starts with a 'p' character.
- After the 'p', a sequence of alphabetic characters terminated by a whitespace provides information about the sub-signal state and it being in input or output state.
- After the whitespace, we find the 0 drive strength for each sub-signal, varying between 0 (weak) and 6 (strong); this section is terminated by a whitespace.
- Similarly to the 0 drive strength, we find the 1 drive strength for each sub-signal, terminated by a whitespace.
- The last value is the gate or bus ID.

Please notice that, in this case, the only difference between a bus or a gate is the number of sub-signals involved: in a bus, we find multiple sub-signals being part of a single signal, while a gate only has 1 sub-signal. Otherwise, changes are treated equally.

In section 2.1.2 we saw an example of a NCSim VCD file. In this case, the main difference with the VCD standard is the usage of the EVCD signal declarations. Moreover, the signal changes do not conform to any standard.

Recently, we included XCellium, also by Cadence [43], among our tools. XCellium provides different VCD flavours, including the IEEE standard VCD [41] and EVCD [42] files. However, EVCD standard files are relegated to small circuits, as a large number of signals overflows the assigned IDs, according to the error messages provided by the software. However, using VCD signal IDs solves the problem. However, in this format, the changes are expressed in the EVCD format.

2.2 The VCD Analyzer

The reason behind the development of this tool is the need to analyze larger and larger VCD files in the least amount of time possible. Moreover, EDA tools provide little to no possibility of performing different analyses at their runtime, which leads to the need of a post-simulation analysis. VCD files generated by long simulations can hold even TeraBytes of data regarding signal changes in the circuit. In particular, the need for this tool was born to perform a toggle activity study on a ST Microelectronics SoC of the SP58 family, which included 20 million gates and 8 million buses.

However, this tool is extensible and performs different types of analysis:

- A Single Point Full Analysis, in which we transform the VCD file into a SQLite [45] database. This is the first analysis introduced, and saves all the status changes of signals contained in the VCD. As VCDs naturally group signal changes by timestamps, we originally wanted to group changes by signal. However, using a SQL database allows us to retrieve all the changes with the desired grouping.
- A Single Point Statistical Analysis, in which we show the toggle coverage among signals, optionally providing information about the duration of each

signal state. Please, notice that the toggle coverage considers only switches between 0 and 1 values of the signal, as x is an undefined state that may be discovered later and z means that the signal is in a high impedance state, not allowing to pass data between various chip components.

- A Multiple Point Couple Analysis, in which we assert a causality relationship on couples of signals. In particular, after detecting the couples that have a causal relationship through the inspection of the circuit, we detect if there is an influence between two or more signals. In particular, we check for couples of signals having opposite values for at least a user-defined amount of time. This leads to understand if there is a causality effect on the signals, which could physically influence each other through electro-magnetic interferences.

To gather as much performance as needed, we focused both on the file reading and on the file parsing areas. Moreover, to provide good performance, the tool is fully written in C++.

2.2.1 File reading

When executing a program, reading a file is one of the slowest parts. This is both due to physical constraints (Storage devices work at a much slower rate than volatile memory) and the C++ standard library being very general-purpose and abstract. In particular, the *iostream* library from C++ is much slower than the C *stdio.h* counterpart, using an object-oriented design and polymorphism for input operations. On one hand, polymorphism ensures that each streaming operation is performed using the same C++ code. However, on the other hand, polymorphism involves a virtual method table (also called *vtable*), which lead to dereferencing pointers that may be anywhere in memory, both resulting in more jump instructions and effectively provoking cache misses due to the jumps. As methods to perform polymorphism not using the *vtable* have been developed [46], C++ compilers do not fully implement them. As for the C library, the *FILE** structure is an opaque type that needs to be dereferenced. Moreover, the function *fscanf* performs itself parsing at runtime, which makes it both versatile and not optimal at the same time.

However, operating systems come to the rescue: the program needs to be run on a Unix-like operating system, such as Linux. Through the usage of the Unix system

calls, it is possible to open a file in binary mode and read it sequentially. However, reading is a synchronous operation, this leads to the reading process effectively halting the execution of instructions on CPU. However, reading operations are expensive, as stated above. Thus, we developed a buffered reader. In the beginning of the development, we had the idea that a double-buffered file reader could have led to better performance, and the interface for the file reader still assumes a double-buffered reading. However, the overhead due to the asynchronous reading resulted much higher than we expected, effectively slowing the reading process compared to the single-buffered file reading. Moreover, even the buffer size may lead to different performance: intuitively, reading 1 KB of data is much faster than reading 1 GB of data. However, there is also a cost to performing a system call, which is needed for reading operations. In our experiments, we found that, even among different hardware configurations, a buffer of around 8 MB of data provides the best performance for our file type.

Another possibility for reading the file could have been memory mapping, which provides a way to perform reading operations on file in the same way as if it was a C array. This leads to the file being loaded in pages, which are usually 4 KB large, and through a clever usage of page faults the operating system manages the memory it can use automatically. However, through our experiments, it resulted in a slower reading of the file. We found the reason for this is a high number of page faults triggered, which effectively need context switches between kernel and user space to be solved.

2.2.2 Pipelined parsing operations

For parsing VCD files, we chose not to use any parser generator. As many of them are available in the market, producing theoretically efficient C parsers [47, 48], they make heavy use of dynamic memory allocation. This does not come as a surprise, as they are general-purpose lexer (the lexical identifier) and parser generators, and are often used to deal with the complexity of creating a language compiler. However, VCD is not a complex standard. Thus, we resorted to manual parsing.

The parsing of the first and second parts and the building of the initial structure of the VCD are performed sequentially, as they are usually the shortest parts of the VCD. However, for the third part, we resort to a pipelined approach.

The pipelined approach provides the entire process being limited only by the slowest stage, as shown in Equation 2.

$$T_{proc} = T_{slowest} \cdot N + \sum_s^{\{all\ stages\} \setminus \{slowest\}} T_s \quad (2)$$

Thus, stages must be balanced to reach a balance. To identify the balance, we define a *pipeline efficiency* P , defined as shown in Equation 2.1. In this formula, the wall-clock time is the time elapsed between the beginning of the pipeline to the end of the pipeline; the sum of times, instead, is the sum of the time spent on each pipeline stage. This efficiency value exists between 0 and 1. The closer to 1 it gets, more the pipeline is balanced.

$$P = \frac{Wall-Clock\ Time}{Sum\ Of\ Times} \quad (2.1)$$

Figure 4.1 shows the ideal pipeline workflow (Figure 2.2a). An unbalanced pipeline is shown in Figure 2.2b, while Figure 2.2c shows that with enough parallelism we can improve the parsing stage, executing it much faster than the slower stages.

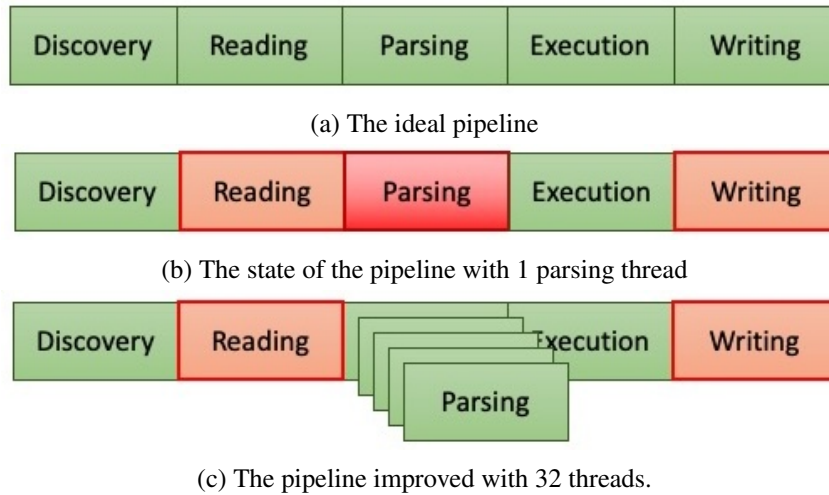


Fig. 2.2 The proposed pipeline with all the stages. Colors represent how fast is a certain stage: green means faster, while red means slower.

In our pipeline, we have 5 stages working concurrently:

- The Discovery stage
- The Reading stage
- The Parsing stage
- The Execution stage
- The Writing stage

Between each stage there is a thread-safe queue for passing data to the next stage. This way, the system is passing data between threads without data racing issues. Between the Parsing stage and Execution stage, however, we may need data to be sorted by generation time. In this case, we use a thread-safe priority queue.

In the Discovery stage, we perform an initial reading of the file and divide it in blocks. In the beginning, the stage was different between the analyses: in particular, in the statistical analysis it did not require the file to be read, and uses a default block size. However, with the support of VCD formatted names, a rare bug could occur where the block of the parsing stage could begin with the character '#'. As the block would begin with it and the parser would interpret it as a timestamp. Thus, the Discovery stage was unified between the analyses, and search for each timestamp. As we will see, the results do not change significantly, as the Reading stage already took much of the time needed for the program execution.

In the Reading stage, we read data in the range defined by the previously defined blocks. To improve performance, in the Reading stage also uses an array of pre-allocated buffered file readers, as allocations and de-allocations slow down the execution of the program significantly.

In the Parsing stage, operations may slightly change between different analyses due to different requirements. However, the common theme is being able to distinguish between changes of signals and timestamps.

In the Execution stage, the analysis operation is performed. Depending on the configuration, however, there may be some exception: in particular, using the statistical analysis, it is possible skip this stage by not detecting the duration of each state.

In the Writing stage, usually performed in the end of the entire execution, the result of the analysis is written back to the file. There is, however, an exception:

as the Full Analysis would saturate RAM for larger files, this is treated as a proper pipelined stage.

2.2.3 Experimental results

In this section, we show the effectiveness and scalability of our tool in terms of computation time and memory usage. In particular, as we focus on multi-thread applications, we report the wall-clock time elapsed by all threads to perform our analysis. We compare these values with the ones gathered from the purely sequential version. The elapsed time is measured with a third-party profiling tool to be as fair as possible. However, the times from the stages are measured with our program itself; this may also lead to small slowdowns in the execution, but it also shows the inner working of the pipeline and its correct implementation.

In our experiments, we limited the maximum central memory to 128 GB to avoid memory being heavily swapped. Moreover, we run each experiment 5 times, and we present average data on all these executions. Tests have been performed on a machine equipped with 2 CPU AMD EPYC 7301, including 16 cores at 2.2 GHz with hyperthreading and 128 GBytes of RAM. All the experiments were performed under the Linux CentOS 7 operating system, using the compiler GCC 11.2 through the package *devtoolset-11*.

Our experimental evaluation is performed on an automotive SoC from the SPC58 family produced by ST Microelectronics [49], using 40nm technology. It has 6 MB of Flash memory and 128 KB of general-purpose SRAM. It contains about 20 million logic gates in the logic parts and about 700 k flip-flops. Therefore, it constitutes a medium-high complexity case study for the proposed. However, this approach is portable to any SoC. This is true for both this section and Section 2.3.6, in which we will talk about a toolchain built around the VCD analyzer.

Full analysis

Tables 2.1, 2.2 and 2.3 show the full time analysis performed on the VCDs of increasing size. Where not specified, the time is reported in seconds.

In particular, in Table 2.1 we show the advantages of a multithreaded environment over the single-threaded one, using an increasing amount of threads for the

#THREADS	DISCOVER	READ	PARSE	WRITE-BACK	TIME SUM	WALL-CLOCK
Sequential	–	–	–	–	–	1793
1	452	172	948	219	1792	1047
4	140	214	289	227	871	329
8	130	176	164	234	687	302
16	390	609	82	128	1211	662
32	124	185	35	215	561	293
64	393	665	22	171	1253	677

Table 2.1 Single point full timing analysis. Running times for the different pipeline stages, with an increasing number of threads, for the smaller VCD file, i.e., the one of 38 GBytes. All times are reported in seconds. The symbol – means that the data is meaningless in that experiment.

VCD SIZE [GBYTES]	SINGLE THREADED TIME SUM	PIPELINE + PARALLEL + OPT					
		Discover	Read	Parse	Write-Back	TIME SUM	WALL-CLOCK
10	355	18	50	9	62	141	83
20	711	35	92	17	109	255	150
38	1792	124	185	35	215	561	293
57	5094	654	735	55	487	2933	916
80	5.50 h	2.06 h	1.92 h	48	107	4.03 h	2.08 h
100	6.95 h	2.65 h	2.47 h	58	151	5.17 h	2.67 h
124	6.40 h	2.92 h	2.72 h	72	295	5.74 h	2.94 h
140	8.39 h	3.59 h	3.34 h	78	245	7.11 h	3.61 h
156	9.21 h	3.71 h	3.45 h	88	285	7.26 h	3.72 h
179	10.30 h	4.66 h	4.34 h	102	435	9.14 h	4.68 h
207	11.77 h	4.58 h	4.26 h	118	423	9.00 h	4.60 h
243	14.12 h	5.50 h	5.12 h	140	618	10.84 h	5.52 h

Table 2.2 Single point full timing analysis. Running times for the different pipeline stages, with 32 threads (our most efficient configuration) for VCD files with increasing size (from 10 to 243 GBytes). All times are reported in seconds or hours (when explicitly stated).

VCD SIZE [GBYTES]	WAITING TIMES		PIPELINE EFFICIENCY	SPEED-UPS	
	Read Stage	Parse Stage		Parse Stage	Entire Process
10	26	16	0.59	12.61	2.80
20	49	14	0.59	12.46	2.72
38	65	24	0.52	26.55	6.10
57	162	616	0.47	36.67	5.56
80	10	2.05 h	0.52	31.12	2.65
100	18	2.64 h	0.52	32.71	2.61
124	36	2.90 h	0.51	28.99	2.18
140	29	3.57 h	0.51	29.87	2.32
156	52	3.67 h	0.51	29.47	2.48
179	54	4.62 h	0.51	28.85	2.20
207	50	4.54 h	0.51	28.99	2.56
243	66	5.44 h	0.51	28.71	2.56

Table 2.3 Single point full timing analysis. In-depth analysis of our approach: Waiting times for the two most critical phases (the Read and Parse stages), pipeline efficiency, and speedups for the Parse phase and the entire process. All times are reported in seconds or hours (when explicitly stated).

parsing operations; the sequential version is not pipelined and provides a baseline for comparing the approaches. Experimentally, we can see that 32 parsing threads provide a sweet spot between the number of parsing threads and the execution time. However, notice that the best pipeline efficiency we get is close to 0.5. This is due to the write-back stage effectively stalling the other operations due to the limited queue space. We will see also in the next results that the disk slowing down operations is a common theme between the analyses, with a different prominence.

In Table 2.2 we show the full analysis performed with 32 parsing threads on files of increasing size, up to 243 GB of data. In those files, we also change the testing program between the file of 57 GB and the one of 80 GB. Below the 80 GB file size, the testing program is generated by an ATPG. For the other files, however, the type of analysis is a functional one, using a real-time operating system (RTOS). This means that there are much less changes between each timestamp, in favour of more fine-grained changes.

In Table 2.3, we show the sum of the waiting times between the most critical stages: the Reading stage and the Writing stage. We see that the Reading stage waits almost no time from the previous stage, while the parsing stage is almost always waiting for the reading stage. We can also see, as previously stated, that the pipeline

efficiency is always close to 0.5. This is why the entire process sees a speed-up of at most around 6x, while the parsing stage provides a speed-up up to around 36x.

Statistical analysis

Tables 2.4, 2.5 and 2.6 show the single-point statistical analysis performed on the VCDs of increasing size. Where not specified, the time is reported in seconds.

#THREADS	DISCOVER	READ	PARSE	WRITE-BACK	TIME SUM	WALL-CLOCK
Sequential	—	—	—	—	—	231
1	0	7	221	0	229	222
4	0	6	63	0	70	63
8	0	8	33	0	41	33
16	0	11	23	0	35	23
32	0	20	20	0	40	21
64	0	45	12	0	58	47

Table 2.4 Single point stress timing analysis. Running times for the different pipeline stages of our chain, with an increasing number of threads for the smaller VCD file, i.e., the one of 10 GBytes. All times are reported in seconds. The symbol — means that the data is meaningless in that experiment.

VCD SIZE [GBYTES]	SINGLE THREADED TIME SUM	PIPELINE + PARALLEL + OPT					WALL-CLOCK
		Discover	Read	Parse	Write-Back	TIME SUM	
10	229	0	20	20	0	40	21
20	470	0	34	34	0	69	36
38	1,132	0	123	45	0	169	126
57	2,656	0	324	129	0	454	327
80	3,634	0	731	149	0	880	735
100	3,985	0	928	218	0	1,146	932
124	5,461	0	1,156	258	0	1,415	1,162
140	5,680	0	1,305	283	0	1,589	1,311
156	4,438	0	1,459	346	0	1,805	1,463
179	1.77 h	0	1,661	391	0	2,053	1,668
207	1.99 h	0	1,867	481	0	2,349	1,875
243	2.49 h	0	2,286	565	0	2,852	2,296

Table 2.5 Single point stress analysis. Running times for the different pipeline stages, with 32 parsing threads (our most balanced configuration) for VCD files with increasing size (from 10 to 243 GBytes). All times are reported in seconds or in hours (when explicitly stated).

In Table 2.4 we show the advantages of a multithreaded environment over the single-threaded one, using an increasing amount of threads for the parsing operations, similarly to Table 2.1; also in this case, the sequential version is not pipelined and provides a baseline for comparing the approaches. Experimentally, we can see that

VCD SIZE [GBYTES]	WAITING TIMES		PIPELINE EFFICIENCY	SPEED-UPS	
	Read Stage	Parse Stage		Parse Stage	Entire Process
10	0	1	0.54	10.90	10.50
20	0	1	0.52	13.25	13.05
38	0	45	0.75	21.85	8.94
57	0	197	0.72	15.72	8.11
80	0	585	0.83	19.46	4.94
100	0	713	0.81	14.07	4.27
124	0	903	0.82	16.81	4.70
140	0	1027	0.83	15.58	4.33
156	0	1118	0.81	9.47	3.03
179	0	1276	0.81	12.12	3.82
207	0	1392	0.80	11.11	3.82
243	0	1729	0.81	11.88	3.90

Table 2.6 Single point stress analysis. In-depth analysis of our approach: Waiting times for all threads of the two most critical phases (read and parse stages), pipeline efficiency P (Equation 2.1), and speed-ups for the Parse phase and the entire process. All times are reported in seconds.

32 parsing threads provide a sweet spot between the number of parsing threads and the execution time. Here we used a smaller file, a 10 GB one, but the proportions remain similar also for bigger files. However, here the pipeline efficiency is at around 0.5 in this small file.

In Table 2.5 we show the statistical analysis performed with 32 parsing threads on files of increasing size, up to 243 GB of data, using the same files that we used in Table 2.2.

In Table 2.6, we show the sum of the waiting times between the most critical stages: the Reading stage and the Writing stage, similarly to Table 2.3. We see that the Reading stage waits no time from the previous stage, while the parsing stage is almost always waiting for the reading stage. We can also see that the pipeline efficiency grows up to around 0.8, showing that a single stage (in this case, the Reading stage) is holding back the speed of the whole execution. The entire process sees a speed-up of around 4x with larger files, while the parsing stage provides a speed-up up to around 10x.

Multiple point couple analysis

Tables 2.7, 2.8 and 2.9 show the multiple-point couple analysis performed on the VCDs of increasing size, similarly to the previous tables. Where not specified, the time is reported in seconds.

#THREADS		PIPELINE STAGES					TIME SUM	WALL-CLOCK
Parse	Execute	Discover	Read	Parse	Execute	Write-Back		
Sequential	Sequential	—	—	—	—	—	—	5884
1	1	15	36	1453	4372	0.0	5877	4385
1	64	17	43	1321	481	0.0	1863	1818
32	32	21	95	64	306	0.0	487	320
32	64	34	135	83	284	0.0	537	309
16	64	29	1451	152	269	0.0	596	292

Table 2.7 Multiple Point Stress Analysis. Running times for the different pipeline stages, with different thread configurations, for the smaller VCD file, i.e., the one of 10 GBytes. All times are reported in seconds. The symbol — means that the data is meaningless in that experiment.

VCD SIZE [GBYTES]	SINGLE THREADED TIME SUM	PIPELINE + PARALLEL + OPT					TIME SUM	WALL-CLOCK
		Discover	Read	Parse	Execute	Write-Back		
10	5887	29	145	152	269	0	596	292
20	5.32 h	60	285	294	481	0	1121	524
38	12.32 h	118	526	564	868	0	2077	957
57	18.23 h	174	684	811	1374	0	3044	1473
80	391.36 h	9.04 h	4.53 h	6.50 h	5.81 h	0	25.89 h	9.02 h
100	524.45 h	10.62 h	5.14 h	7.64 h	6.74 h	0	30.14 h	10.71 h
124	528.31 h	12.76 h	6.34 h	9.15 h	7.96 h	0	36.21 h	12.84 h
140	587.56 h	14.56 h	7.55 h	10.54 h	9.77 h	0	42.42 h	14.67 h
156	716.79 h	15.61 h	7.41 h	11.20 h	9.68 h	0	43.91 h	15.70 h
179	801.82 h	17.72 h	8.39 h	12.70 h	11.06 h	0	49.87 h	17.80 h
207	957.45 h	19.86 h	9.81 h	14.21 h	11.79 h	0	55.67 h	19.94 h
243	1076.56 h	23.65 h	11.14 h	16.97 h	15.06 h	0	66.82 h	23.74 h

Table 2.8 Multiple Point Stress Analysis. Running times for the different pipeline stages, with the 16/64 threads configuration (the most efficient one in our experiments) for VCD files with increasing size (from 10 to 243 GBytes). All times are reported in seconds or hours (when explicitly stated).

In Table 2.7 we show the advantages of a multithreaded environment over the single-threaded one, using different amount of threads for the Parsing and Execution stage; the sequential version is not pipelined and provides a baseline for comparing the approaches. Experimentally, we can see that 32 or 16 parsing threads provide a sweet spot between the number of parsing threads and the execution time. However, the rule of thumb for the execution stage is "the more threads you feed it, the better it is". We capped it at 64 threads, but for many couples it can go as high as the number of CPUs in the host system.

VCD SIZE [GBYTES]	WAITING TIMES		PIPELINE EFFICIENCY	SPEED-UPS	
	Read Stage	Parse Stage		Execute Stage	Entire Process
10	72	128	0.49	16.24	20.12
20	166	218	0.47	32.59	36.51
38	335	381	0.46	42.68	46.29
57	684	653	0.48	37.27	44.53
80	4.33 h	2.61 h	0.35	42.32	43.13
100	5.26 h	3.06 h	0.36	45.45	49.32
124	6.13 h	3.67 h	0.35	39.61	41.76
140	6.69 h	4.09 h	0.35	38.78	40.93
156	7.83 h	4.47 h	0.36	44.33	46.23
179	8.90 h	5.07 h	0.36	42.96	45.62
207	9.55 h	5.69 h	0.36	45.46	48.73
243	11.9 h	6.73 h	0.36	44.89	45.98

Table 2.9 Multiple Point Stress Analysis. In-depth analysis of our approach: Waiting times for all threads of the two most critical phases (read and parse stages), pipeline efficiency, and speed-ups for the Parse phase and the entire process. All times are reported in seconds or hours (when explicitly stated).

In Table 2.8 we show the multiple-point couple analysis performed with 32 parsing threads and 64 execution threads on files of increasing size, up to 243 GB of data with 2'000'000 couples. Those are the same files used for the previous analyses. In this analysis, the execution stage takes care of analyzing the couples and detecting if they have been in opposite values for a certain time. Moreover, the discovery stage performs heavier calculations to lower the pressure on the execution stage, as it divides the file based on the timestamp difference. Although we were not completely satisfied with this balance, we found this combination was the best trade-off we could come up with. Thus, in this case, the discovery stage is the bottleneck of the operation. However, this is due to few couples being analyzed. The execution time grows linearly with the number of couples to analyze. Using 10 times the couples, the bottleneck becomes the execution stage.

In Table 2.9, we show the sum of the waiting times between two critical stages: the Reading stage and the Parsing stage. We see that the Reading stage waits a significant amount of time from the previous stage, and the Parsing stage is almost always waiting for the reading stage. We can also see that the pipeline efficiency is around 0.36 for larger files. However, the entire process sees a speed-up of at around 45x, while the parsing stage provides a speed-up up to around 45x for larger files. This is due both to the pipeline approach and the Execution stage being split

between multiple threads. Moreover, since the Execution stage analyzes every couple independently, the problem can be massively parallelized. However, as we tried to make it work on a GPU, we found that the number of signals and changes that need to be passed between CPU and GPU is so large that the data transfer takes up most of the time of the execution stage.

2.2.4 Conclusions

With this methodology, we were able to analyze large VCD files in a small amount of time. As circuits become larger, the single-point approaches scale linearly with the duration and size of the chip. However, the multiple-point analysis may need more and more computational power depending on different factors: the number of multiple-point subjects to be analyzed (in our case, couples), the complexity of the analysis and the file size.

A common issue is that the amount of time spent is also dependent on the program that generates the changes. A lower number of changes for each timestamp leads to increased time for the analysis, and this is especially true for the Execution stage of our couple analysis, since it works with precise timestamps to gather the durations needed.

However, the VCD tool and, in particular, the statistical analysis, are now part of a toolchain we developed for the analysis of larger programs.

2.3 The VCD Toolchain

This section follows this paper [20], also giving as granted the previous section for what concerns the VCD tool.

While the VCD tool started being used, the need for support tools raised. Since a first step was available. This led to a standardization of the output file format, together with the creation of tools aimed to analyze the results of the VCD analyzer. In particular, the following tools were developed:

- A layout-aware analysis tool to weight the results.

- A set-theory based tool to compare and potentially merge more VCD results; it will be called, from now on, the *set tool*.
- A tool for the visualization of the chip with its coverage.
- A tool for computing sub-coverages among different modules.

Those tools became parts of a toolchain, depicted in Figure 2.3. With this toolchain, we are able to build a funnel that starts from the chip simulation and ends with the desired results, both visually and with formal information about the desired modules under test.

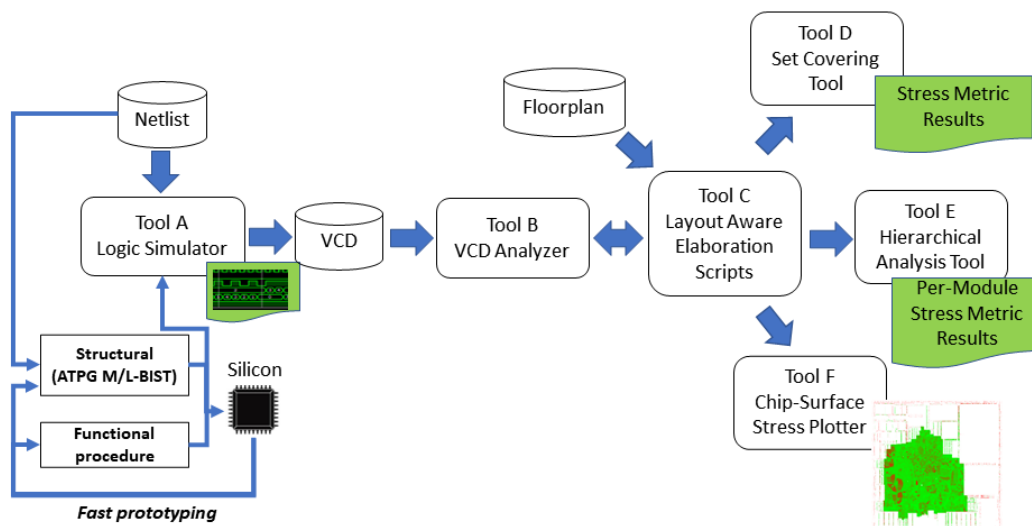


Fig. 2.3 The toolchain built around the VCD

2.3.1 The standard file

We designed the toolchain in such a way that steps can be executed interchangeably. This is because we may not know if we need any tool from the start, and we might even exclude some tools from the analysis if needed. For instance, if we only have one test, the set tool does not provide interesting insights. Thus, we needed a file that has all the information needed for every step.

Luckily, the set tool is the most avid of information, mostly due to its merging capabilities. In our file, we can find the following data collected for each signal:

- The name of the signal.
- The ID and sub-ID of the signal, to trace it back to the VCD.
- Its statistical toggle coverage (0.0, 0.5 or 1.0).
- Whether the signal toggled from 0 to 1.
- Whether the signal toggled from 1 to 0.
- The initial value of the signal, which is important for the result merging operation.
- The times that the signal spent at states 1, 0, z and x, together with the relative percentages over the total time.

The last information is somewhat new, however, and not handled by the following tools, as it did not yet prove useful to them.

2.3.2 Layout-aware analysis

In this phase of the proposed toolchain, a set of scripts is included to provide layout-awareness capabilities. This toolchain elaboration step is exploited to link the stress evaluation with the actual SoC physical characteristics.

The sub-steps are summarized in Figure 2.4, and three main components can be individuated:

- Virtual node elimination: it eliminates virtual nodes and signals that do not have a physical implementation.
- Topology analysis: it analyzes the floor plan of a given SoC to generate helpful information on the neighbor gates used for multi-point stress metric analysis in the VCD file analyzer.
- Metrics Weighting: the stress metrics can be weighted using a bi-dimensional density by exploiting layout information.

In the following subsections, I will go into details on every component of the Layout-aware elaboration analysis tools.

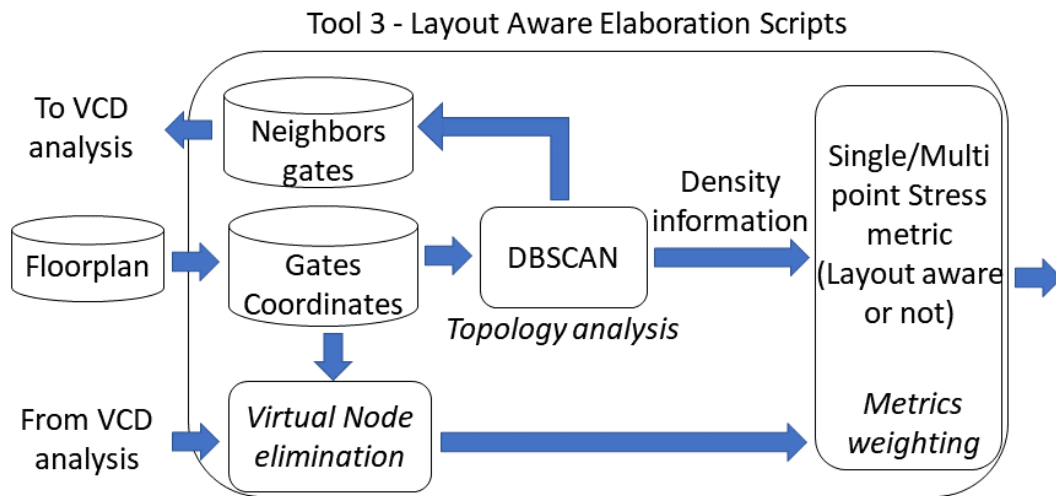


Fig. 2.4 The workflow for the layout-aware analysis

Virtual Node Elimination

A significant issue in the stress coverage evaluation process comes from the pure VCD analysis of the simulation dump. As a matter of fact, by looking at the VCD header part, the number of signals saved in the dump includes many replications. As shown in Figure 2.5, a circuit connection traversing hierarchies is memorized several times in the VCD.

In particular, the path to and from the not gate ports includes physical circuits points a to d and f to i . However, the VCD dump also includes points b , c , g , and f , which are not corresponding to any real circuit point but are inherited as simulation artifacts. Including extra points leads to longer computations and affects the stress metric value because an unexcited gate may reflect in many VCD signals, thus polluting the final stress metric value.

In order to eliminate the excess of information given by the artifact and not affect stress metrics, we use a layout-aware tool is used to filter signals in excess. Such a method is similar to a collapsing strategy, however it is not based on a netlist analysis, but on the layout information, as shown in Algorithm 1, only preserving the signals that we find in the layout.

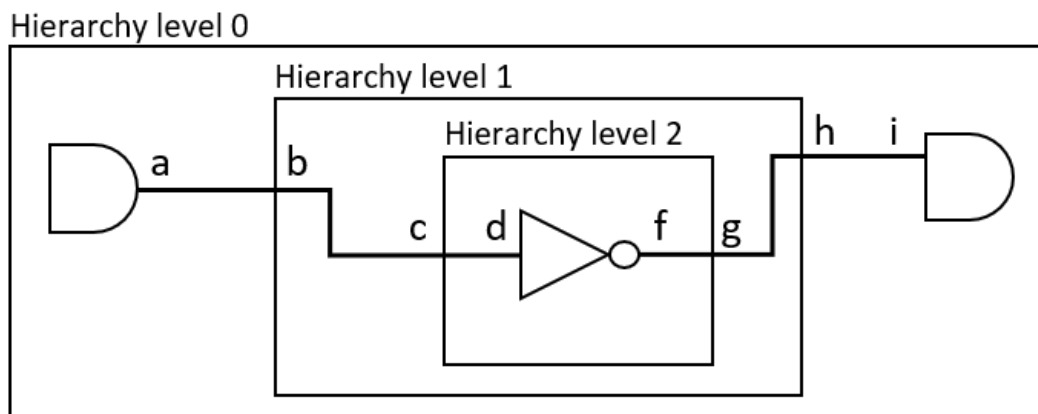


Fig. 2.5 An example of the circuit as described in the VCD file.

Algorithm 1 Virtual node elimination

Require: Layout information, VCD file

Read and save physical gates into a hash table.

for signal in VCD **do**

if signal in the hash table **then**

 Preserve the signal.

end if

end for

Topology Analysis

When dealing with a complex System-on-Chip, it is fundamental first to understand its topology, i.e., how gates are physically placed across the layout to gather valuable insights that can help understand the meaning of the computed stress metrics and devise tests to cover all the parts of the SoC adequately and uniformly.

Modern SoCs do not show a uniform distribution of gates on the layout front-end [35]. Traditional stress metrics are usually *gate-based*, as they consider the behaviour of a gate or a set of gates regardless of how the SoC is structured. Moreover, they are *unweighted*: they consider each gate to have the same contribution to the final metric. However, the stress per unit of the area varies across the layout, and it may lead to different aging scenarios depending on the density of gates in a given area. For instance, a more dense area can lead to faster aging, due to the heat being propagated among neighbour gates. Thus, knowledge of the device topology is crucial to assess the quality and uniformity of the stress imposed by a test. Thus, we need to cross information between the layout and the VCD analysis. This allows us to:

- Introduce gates aggregation when measuring multiple-point stress metrics.
- Reach a high level of accuracy by weighting the stress measurement according to the SoC density, individuating critical areas on the SoC.
- Generate SoC stress heatmap plot by providing information about gate coordinates.

It is essential to mention that exhaustive methods exist for computing the 2D density. However, when we use exhaustive methods, the computing time grows with the dimension of gates, making the exhaustive methods unfeasible for larger SoCs. In this case, to overcome computing time limitations, we use a very well known Machine Learning algorithm: the *Density-based spatial clustering of applications with noise*, or DBSCAN [50]. The idea of DBSCAN is to generate a set of clusters, grouping data in the same group with some similarities, aiming to divide the good data from the outliers. DBSCAN is a density-based clustering algorithm. Given a set of points, it groups closely packed points (points with many nearby neighbors), highlighting outliers in the low-density region. As DBSCAN distinguishes the "good

data" based on closeness and a configurable threshold, we are able to gather different density levels through the repeated use of the algorithm.

With a fixed inter-gate distance, all the logic gates of the SoC can be organized into sets of neighbors for performing multiple-point analysis. In the toolchain flow, a gate pair located at a distance smaller than a selected threshold on the layout is considered of interest and extracted to feed the VCD analysis tools for the multiple-point analysis.

However, the computation time costs to extract a couple of gates close enough to each other are traded off with accuracy; a classification method that implements clustering is proposed to abate the timing costs while guaranteeing sufficient accuracy in the selection. In particular, the clustering technique is based on the individuation of clusters of nodes, where every cluster contains the nodes located in a sufficiently dense area of the layout.

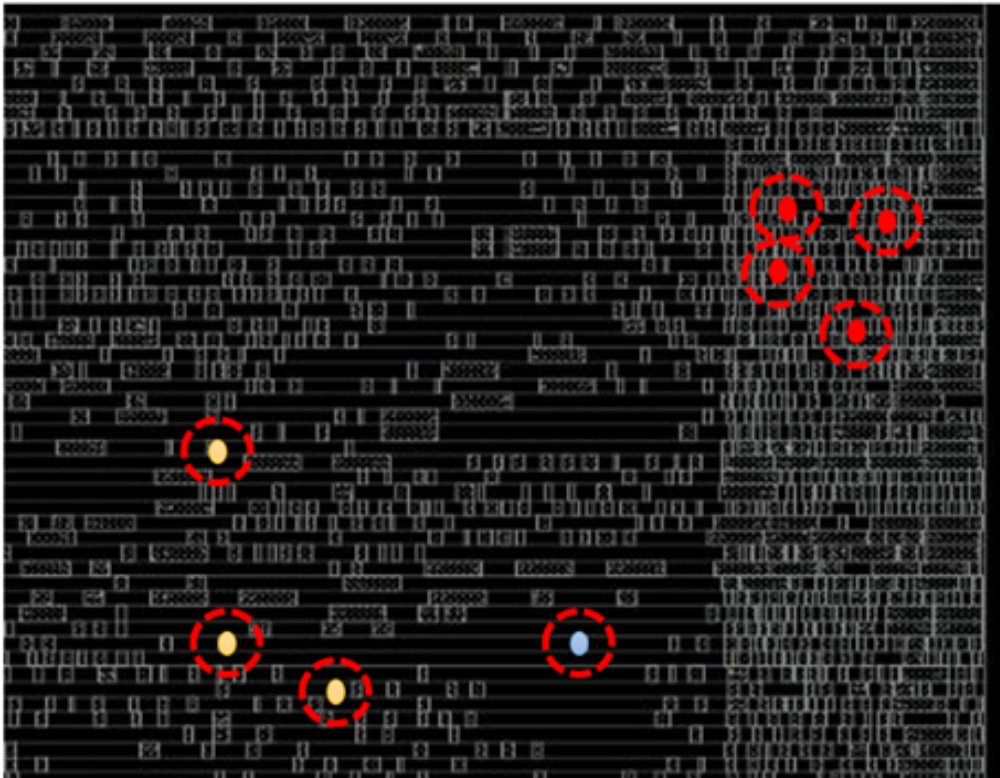


Fig. 2.6 An example of the clustering method (DBSCAN) executed on a generic layout

Figure 2.6 illustrates with an example how the algorithm works on a generic front-end layout: The classification process divides SoC gates into core points (in red), border points (in yellow), and noise points (in blue), and it analyzes the neighborhood within a fixed inter-gate distance (red dashed circle). Once the clusters are identified over the SoC surface, the list of couples (or sets) can be performed in a little computation time. Furthermore, the method is helpful for successive steps of the flow, particularly for weighting the stress activity based on the SoC density, which can vary from one region to another in the SoC layout.

Metrics Weighting

As mentioned in the previous subsection regarding the topology analysis, the stress per unit of area differs across the layout in modern SoC. Depending on the density of gates in a given area, it may lead to different aging scenarios. Therefore, knowledge of the SoC topology is crucial to assessing the quality and uniformity of the stress and enhancing the stress metric with density awareness as proposed in [35]. This sub-step of the toolchain provides formulas and considerations presented in [35].

Consequently, a layout-aware stress metric is crucial for both the single and the multiple-point stress metrics to weigh the stress over denser areas of the SoC instead of considering all the gates equal among them.

2.3.3 The Set Tool

During the BI, different stress approaches of various natures can be used. For example, a BI phase can run scan-based patterns, logic or memory BISTs procedures, and functional programs. Distinguishing which coverage is granted by which method (i.e., which pattern covers a specific section of the SoC) is crucial, as it allows an understanding of which approaches provide more remarkable improvements.

Therefore, the proposed toolchain includes a tool for implementing a set covering analysis. This tool receives the stress results collected by adopting patterns of different natures, generating all possible set interactions, and providing insights about the stress percentage for each stress approach. Moreover, it is also able to merge the results of subsequent analyses.

In particular, the set tool provides the following features:

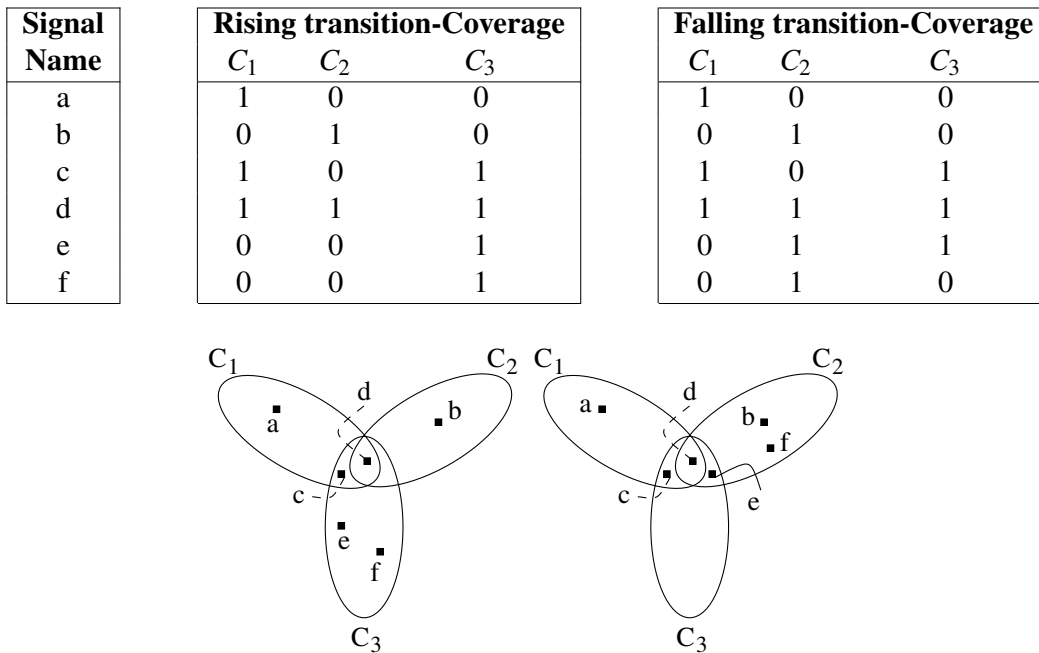


Fig. 2.7 Rising and falling transition set cover: from file to set interacts

- A confusion matrix of toggle coverages among different stress approaches.
- The list of unique toggle coverages for each stress approach.
- The number of times that a signal toggles.
- The possibility to merge different results, using the different coverages as a subsequent superimposition of stress approaches.
- The possibility to identify raising or falling transitions of a given gate for each stress approach.

In this tool, the main challenge was lowering the memory footprint while also performing a parallel analysis among the signals provided by different files. To lower the memory usage, we exploited the fact that signal names are unique and immutable and tend to be referenced by many files. This allowed us to use a technique called *string pooling* or *string interning* to avoid having equal strings in memory, as strings tend to have the largest memory usage in the program (a Rust String object consists of: 8 bytes for the pointed memory, 8 bytes for its capacity and roughly 1 byte per character located in the heap). Using string pooling, signals in different files share references to their names instead of having more instances of the same string.

However, reading the input files is the slowest part of the process, string pooling is enabled when moving the signals to an appropriate data structure; in this way, we avoid slowing down the file reading stage, as string pooling requires a mutex to avoid data races among threads. In the beginning, the application reads all covering sets directly from the file and stores them in hash tables using the identifiers of the signal as a hash key. Later, as we identify the sequence of signals, we move them to a dynamic array.

Figure 2.7 show an example of the entire process. The upper table shows the original set covering representation, which stores each rising and falling transition covering for each signal. The bottom Venn diagrams show the set interpretation and highlighting per-pattern subsets.

Suppose to have N signals (with N potentially very large) and M set covering (with M limited by the number of different covers), the application has an $O(N \cdot M^2)$ time complexity and memory complexity. As the value of M is usually limited to a few units, the time required by the entire process is restricted to a few tens of seconds in the worst case. Moreover, using a single bit for each signal value reduces the memory usage to a few GBytes.

2.3.4 The Hierarchical Analysis Tool

The *divide et impera* approach in digital design has increased the overall complexity of SoCs, allowing teams to focus more on a design of a single entity instead of the whole SoC. Following this approach, today's SoCs comprise several subunits with a variable number of other nested subunits up to the leaves, i.e., the logic gates. Therefore, considering metrics on SoCs with millions of gates, it becomes evident that coarse metrics on the overall device have less meaning than thorough computed metrics on modules below the average coverage of the whole SoC.

The stress analysis frequently focuses on some specific module, usually the ones showing low coverage. To support this analysis, the flow includes a selection process implemented as a tool that extracts critical modules below a given threshold and a per-module stress coverage.

The hierarchical analysis tool analyzes a stress pattern, or a set of them, to produce a module-based coverage file where the module and its sub-modules have their coverage coupled together with their design hierarchy.

The hierarchical analysis starts from the standard human readable text-based input file of coverage described in Section 2.3.1. Table 2.10 presents an example of a coverage file, where signal names contain the hierarchy with the associated stress coverage.

Signal Name	...	Coverage	...
a/b/c	...	1.0	...
a/b/d	...	0.5	...
a/b/f/g	...	0.5	...
a/h	...	1.0	...

Table 2.10 A simplified view of the coverage file, highlighting the information looked by the hierarchical analysis tool

The tool is independent of the SoC and analyzes the input file sequentially to avoid non-deterministic access to data structures. Consequently, it analyzes only the coverage file recreating the hierarchy by decomposing the path. In other words, using as an example the design represented in Table 2.10, the top-level unit *a* is decomposed into a subunit called *b* and *h*; *b* is further decomposed into its children, i.e., *c*, *d*, *f*, and the leaf *g*.

Internally, it works on parsing strings in such a way as to create a tree of modules and sub-modules, starting from the top entity; for each node of the tree, it calculates and saves the list of signals and their coverages in the internal data structure. The tool creates the tree-coverage structure depicted in Figure 2.8.

An important aspect to mention is that the coverage of a general parents node in the tree hierarchy follows the recursive formula:

$$Cov(node) = \frac{\sum_{i=0}^{\#children(node)} Cov(i)}{n} \quad (2.2)$$

Where *n* is the number of leaves in the sub-tree, if the number of children of the node is 0, meaning the node is a leaf, then the node coverage is returned. The formula can generate, given a module, its related stress coverage by computing the average on all the children nodes.

In particular, the tool generates warnings on those modules in the hierarchy that does not reach the acceptable level of coverage, given as an input parameter. Moreover, it produces a file containing every module with its associated level in the

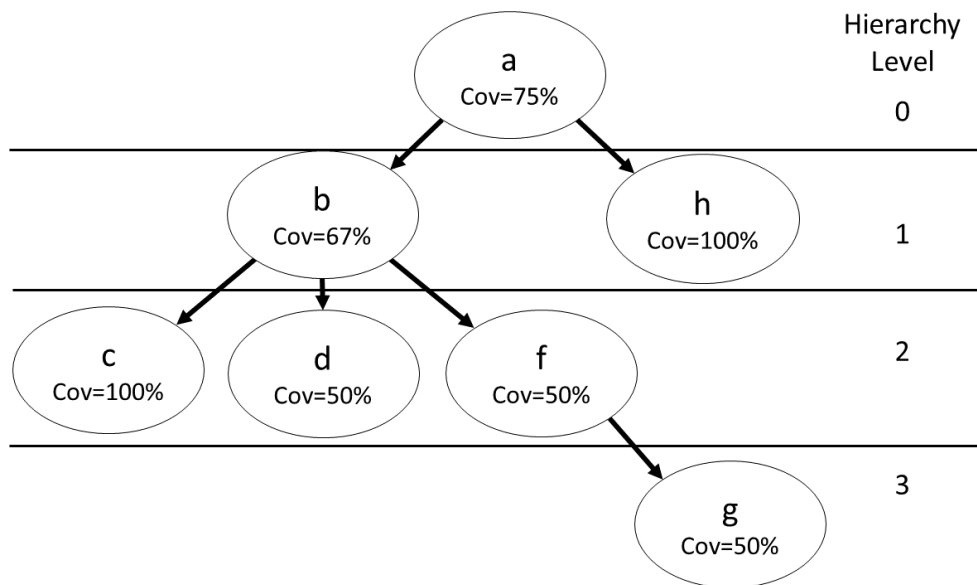


Fig. 2.8 A high-level view of the tree data structure containing the coverage for each node

hierarchy and the related stress coverage. Instead of searching for a not satisfied coverage module, the tool also accepts a module name (a string) to extract the stress coverage and generate a coverage file within the module hierarchy.

A hierarchical decomposition of SoC modules eventually allows for test engineers to focus on the desired module, which would more likely give a more significant step up into the coverage than modules that have already reached an acceptable level.

2.3.5 The Chip-Surface Stress Plotter

Although an essential aspect of the stress approach is a quantitative information provided by the hierarchical analysis tool and the set covering tool, a qualitative visualization of stress plays a crucial role in understanding which modules lack testing effort. Qualitative visualization of stress over the SoC layout allows locating stress pattern weaknesses and easily highlighting the coverage abilities of different stress patterns.

The plot tool uses as input the standard file described in Section 2.3.1 to generate a heatmap of the stress over the SoC layout.

Figure 2.9 presents an example of such a heatmap over a generic SoC.

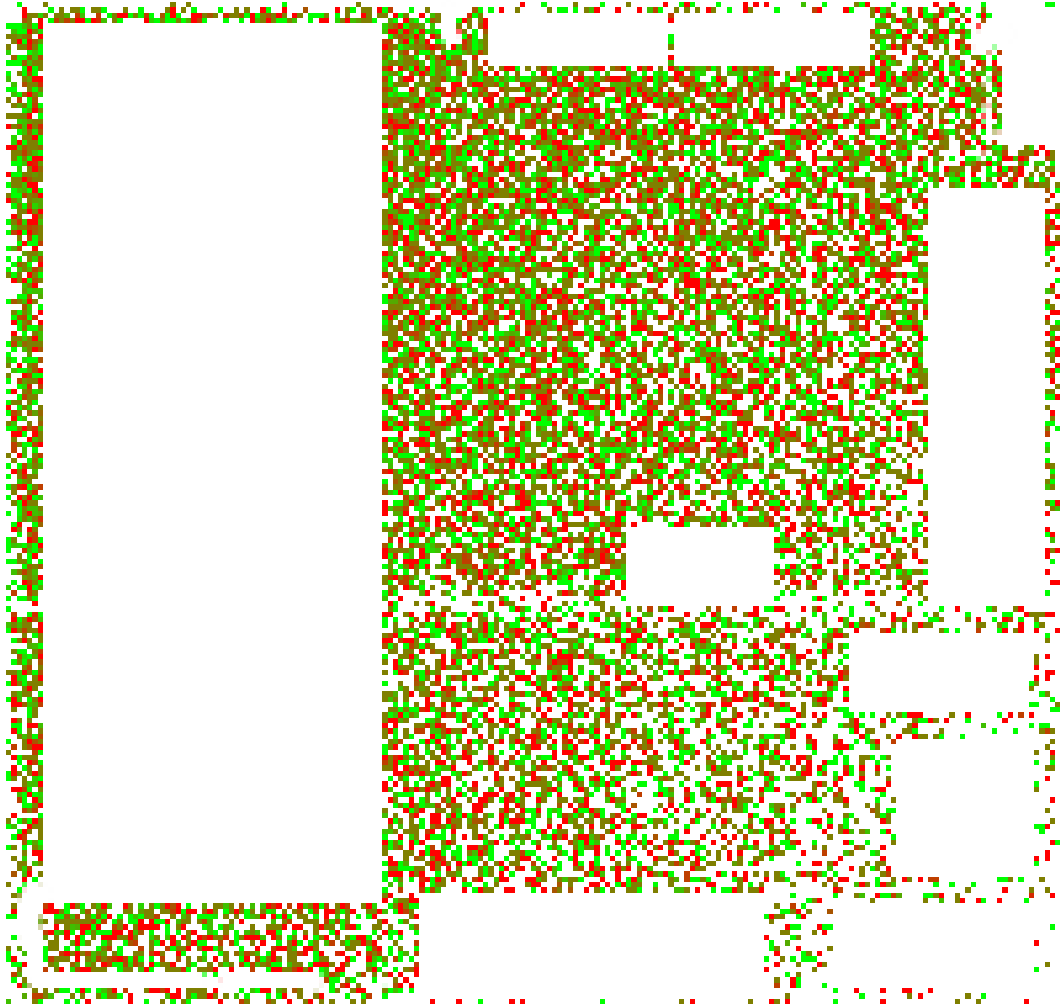


Fig. 2.9 An example of stress heatmap over a generic SoC layout

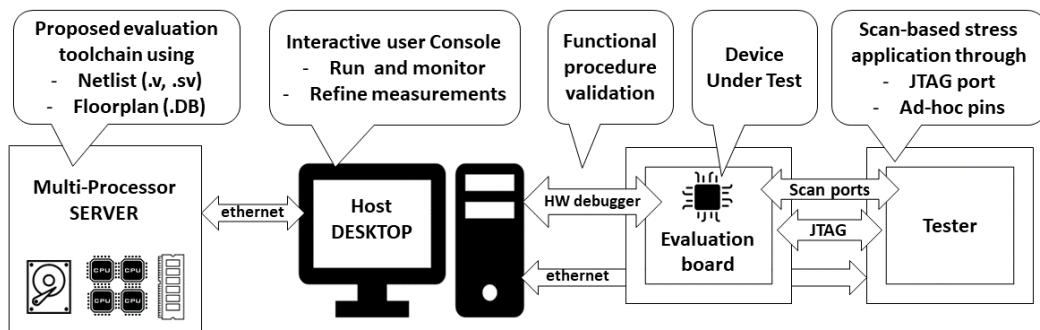


Fig. 2.10 Experimental setup

All signals are mapped into pixels that summarize the stress applied to the device gates, while white zones are embedded memories that are not part of the BI grading. Gate stress coverage resides between 0% (red pixel) and 100% (green pixel), with values in the middle represented by a color gradient.

As it can be seen from Figure 2.9, there exist portion of red that correspond to non-stressed modules, portions of green (stressed modules) in different shades depending on the resolution of the image, and islands of white that correspond to memories, analog and power modules (outside the scope of BI).

Since images of arbitrary resolutions can be generated, it is important to speed up the drawing process depending on the level of detail we need in parallel. GPUs are usually the way to go for image elaboration, as their parallel computation capabilities vastly surpass the ones of the CPU. However, to guarantee compatibility with every device, we perform computations on CPU, resorting to the widely used SDL2 libraries [51] for saving the image into the standard BMP format. SDL2 libraries provide APIs for image creation and manipulation on the CPU and GPU. It also provides methods to color the image pixel-by-pixel. Since every pixel is independent, this tool assigns each pixel to a different thread. In this case, we use OpenMP [52], a widely used library for easing parallel patterns implementation to perform image generation. Moreover, as we may not need a precision based on cells, the tool is also able to generate smaller images with average data between neighbour cells. This way, even with larger chips, it is possible to conceptually *zoom out* the image and have a faster generated heatmap.

2.3.6 Experimental results

In this section, we use the same setup as in Section 2.2.3. However, here we will focus on real tests developed for the chip, in opposition to the previously mentioned section, as we want to perform an experimental evaluation on a real case. The target device is a 40 nm Automotive SoC [49] belonging to the SPC58 family manufactured by ST Microelectronics and compliant with the standard ISO26262 ASIL-D. In the following, it is referred to as DUT. The DUT has a multi-core architecture with three 32-bit cores using the PowerPC Variable-Length Encoding (VLE) instruction set. It has 6 MB of Flash memory and 128 KB of general-purpose SRAM. It contains about 20 million logic gates in the logic parts and about 700 k flip-flops, as mentioned in

Section 2.2.3. Due to its characteristics, it constitutes a medium-high complexity case study for the proposed toolchain. As far as the stress flow during the BI phase is concerned, the DUT is stimulated by:

- A configurable scan chain stimulates the DUT from regular pins using scan-based patterns.
- A logic and memory Built-In Self-Test (BIST) activated from inside or outside the device.
- Some functional programs are executed by the DUT.

To perform our analyses, the RTL and gate-level description of the DUT, as well as the layout, must be known. Furthermore, the manufactured DUT must be available to speed up the development process of functional and structural stress approaches [53].

Acronym	Stress Pattern	Execution time	Exhaustive simulation	Deductive simulation	VCD file size
32 ATPG	32 ATPG scan patterns	2.5 ms	5,376* h	45 m	8.4 GB
12 ATPG+	12 ATPG scan patterns (additional)	2.0 ms	4,122* h	39 m	4.0 GB
1024 PR	1,024 Pseudo-Random scan patterns	7.0 ms	172,032* h	567 m	251 GB
LBIST	Logic BIST	3.0 ms	14 h	N/A	182 GB
MBIST	Memory BIST	52 ms	240 h	N/A	3,300 GB
FUNCT	Functional (RTOS boot)	2.1 ms	8.5 h	N/A	184 GB

Table 2.11 The logic simulation phase: CPU times and size of the VCD files generated. The symbol “*” means the time is estimated. “N/A” means that the value is Not Available.

2.3.7 Experimental setup

Figure 2.10 illustrates our simulation and analysis setup designed to evaluate BI stress effectively and mitigate the computational costs of the evaluation. The experimental setup includes two different phases on different platforms:

- A development phase. In this phase, we use the manufactured SoC to run and verify quickly structural and functional stress patterns. A average-performance desktop computer executes the local stress pattern validation phase by communicating with the evaluation board and an ad-hoc developed tester [54, 53]. Our computer is equipped with a quad-core Intel Core i7 running at 2.8 GHz, and 16 GBytes of main memory.

- An analysis phase. In this stage, we evaluate the structural and functional stress patterns. We run the toolchain, from the simulation phase to the extraction of final BI metrics, on a high-performance multi-processor server. The server has an Intel Xeon Gold 6238R processor with 112 CPUs. These processors have 64 bits architecture, allow two hardware threads per CPU, and run at 2.2 GHz. Moreover, the system has 256 GBytes of RAM, a storage system of 8 TB, and a network disk of an additional 10 TBytes. The operating system orchestrating the server is CentOS Linux 7.

We use the manufactured SoC during the developing phase to boost the validation phase with stress patterns coming from different sources, such as ATPG (using single or multiple scan ports), LBIST engines (using compressed test patterns), firmware controlled MBIST engine and pure functional programs. The validation step is supported by a tester, which applies scan-oriented stress, whereas a hardware debugger supports the development of functional procedures.

For example, validating a stress pattern requires a few seconds (functional stress pattern) to minutes (structural stress pattern), depending on the type of pattern to be executed on the manufactured SoC. More in detail for the time-consuming structural stress patterns, during this phase, they may be applied by resorting to a low-cost micro-controller [54] or a more complex tester based on a Zynq Ultrascale+ MPSoC ZCU104 evaluation board by Xilinx [53]. The application time of structural stress patterns is higher than functional stress patterns due to a large amount of provided data to the high number of scan cells (around 700k); the low-cost electrical connections impact the maximum application frequency of structural patterns.

Downstream of the developing phase, the evaluation process starts on the server, where it must guarantee enough resources to:

- Store huge files on disk, up to Terabytes, generated by the initial simulation and then post-processed to collect refined results.
- Keep in the main memory all required information, up to tens of Gigabytes while running the post-process phase.
- Distribute parallel tasks to CPUs.

2.3.8 The Logic Simulator

The logic simulator is the first step in the proposed toolchain. It is based on a commercial logic simulator capable of generating a VCD file, which stores all the signal events during the logic simulation.

As mentioned, during the BI phase, the DUT is exposed to structural and functional stress patterns. Therefore, the logic simulator phase can apply structural and functional stress patterns to the DUT and dumps the related VCD file. The simulation of a structural or a functional stress pattern is substantially different.

In the functional case, the simulation exactly reproduces the behavior of the functional test program. Therefore, a functional test program should be as short as possible from the perspective of execution time.

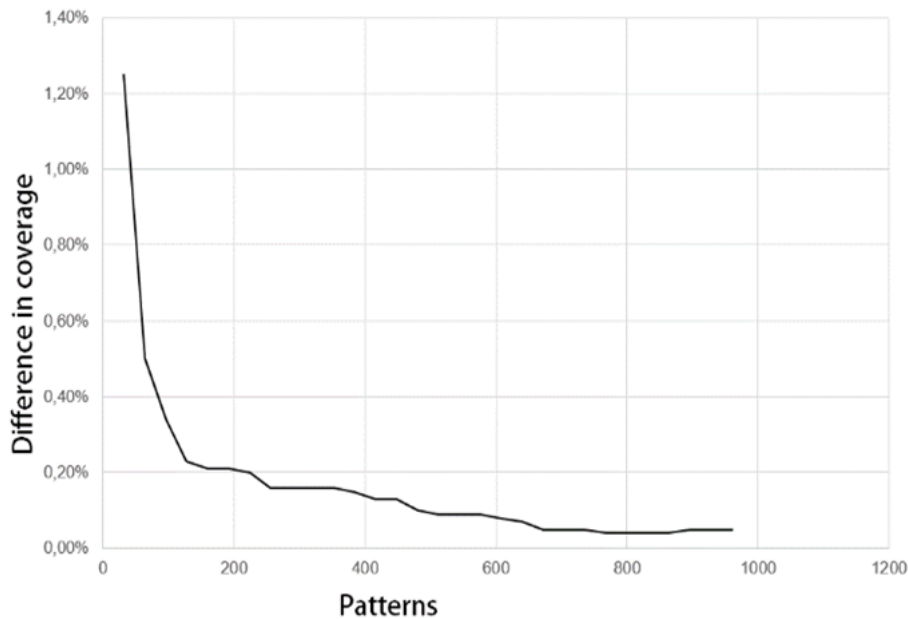


Fig. 2.11 Coverage difference between exhaustive and deductive structural simulation for OpenRisc 1200.

Regarding the structural patterns logic simulation, as already proved in [35], the exhaustive simulation of a complete shift of the scan chain may require an extremely high number of clock cycles. Exhaustive simulations have been executed on the open-source benchmark OpenRisc 1200, which is small enough to allow the exhaustive simulations to be performed. When 200 patterns are applied, the full results show that the difference in terms of stress coverage between the two strategies amounts

to less than 0.2% as Figure 2.11 shows, whereas the difference in terms of time is, on average, around 1700x more for the complete simulation, and it depends on the number of applied stress vectors. Therefore, the deductive simulation has the tradeoff of a slight decrease in the final coverage but strongly affects the computing time.

Following this strategy, Table 2.11 shows the simulation time for all stress patterns considered in the given case study. Case in point, the simulation of a 32 ATPG stress vector lasts about 45 minutes using our deductive approach, whereas the estimated time required to run the exhaustive approach would be about 5,376 hours (i.e., seven months). These data prove the validity of the deductive approach, which, as expected, can provide precise and conservative results with reasonable computational effort.

In Table 2.11, simulation times for the deductive approach regarding Logic and Memory BISTs and functional test programs are not calculated since they are not based on shifting patterns along the entire scan chain.

2.3.9 The VCD File Analyzer

This tool is the tool described in Section 2.2. The VCD analysis performs a transformation of the original VCD file from a time-based view to a signal-based view. Experimental results presented in [15, 16, 18] are updated with VCD files ranging from a few GBytes to TBytes. The execution time bottleneck, as stated in [18], is always the file reading from the disk.

Input VCD	File size	Execution time	Out File size	Rt Memory Usage
32ATPG	8.5 GB	51 s	3.2 GB	11.4 GB
12ATPG+	4.0 GB	18 s	3.2 GB	11.4 GB
1024PR	251 GB	38 m 50 s	3.2 GB	11.4 GB
LBIST	182 GB	21 m 52 s	3.2 GB	11.4 GB
MBIST	3300 GB	10 h 37 m 27 s	3 GB.2 GB	11.4
FUNCT	184 GB	28 m 58 s	3.2 GB	11.4 GB

Table 2.12 Profiled execution for the VCD Analysis for the single-point stress metric.

Table 2.12 shows single-point analysis elaboration times for each stress pattern; the size of input and output files is reported. Main memory occupation is

constant between the stress patterns, as each signal contains only trivially copyable elements [55] regarding the analysis; thus, memory occupation only depends on the signal names and the number of signals themselves. On the other hand, the output file size is always constant and directly proportional to the number of signals in the VCD file.

On the other hand, in Table 2.13, elaboration times for multi-point analysis of different stress patterns are presented, with input and output file sizes and runtime memory usage.

Input VCD	File size	Execution time	Out File size	Rt Memory Usage
32ATPG	8.5 GB	4 m 19 s	307 MB	11.4 GB
12ATPG+	4 GB	3 m 5 s	307 MB	11.4 GB
1024PR	251 GB	4 h 47 m 45 s	307 MB	11.4 GB
LBIST	182 GB	4 h 36 m 44 s	307 MB	11.4 GB
MBIST	3300 GB	63 h 1 m 40 s	307 MB	11.4 GB
FUNCT	184 GB	42 h 51 m 4 s	307 MB	11.4 GB

Table 2.13 Profiled execution for the VCD Analysis for the multi-point stress metric.

As Table 2.13 shows, the runtime memory usage is constant across different stress patterns due to the internal data structure containing the signals. The output file size is reduced since the couples are saved as an incremental, unique index. Regarding the execution time, experimental results in Table 2.13 depict the advantages of resorting to parallel programming for analyzing multi-point metrics in a reasonable amount of time.

2.3.10 Layout-Aware Elaboration Scripts

In this phase of the proposed toolchain, a set of scripts provides layout-awareness capabilities to the entire toolchain. Consequently, the following experimental results depend on the analysis of the DUT layout. In the following subsection, experimental results on the Layout-aware elaboration scripts are presented in detail.

Virtual Node elimination

The layout information is extracted, in advance, from the physical design of the DUT, generating a file that contains the physical positions of logic gates within the layout.

The aforementioned preliminary step allows refining the stress metric produced by the VCD analysis tool in such a way as to correctly consider only physical, implemented logic gates.

As time and memory usage depend mainly on the output size of the VCD elaboration the result, this step has a fixed cost (proportional to the number of signals present in the input file) for each analyzed DUT as Table 2.14 shows.

Stress Pattern	Input File Size	Output File Size	Execution Time	Runtime Memory Usage
1024PR	3.2 GB	2.3 GB	365 s	16.9 GB
MBIST	3.2 GB	2.3 GB	365 s	17.0 GB
FUNCT	3.2 GB	2.3 GB	401 s	17.0 GB

Table 2.14 Profiled execution of Virtual node elimination script.

Topology analysis

The topology analysis on the layout of the DUT is performed as a preliminary step of the toolchain to generate all the required data.

The DUT layout used as a case study can be seen in Figure 2.12, where an important concept can be highlighted: typically, an SoC does not show a uniform distribution of gates. The differences in the gate density distribution of the various parts of an SoC are further highlighted in the Figure 2.12; a brighter shade of green describes parts with a higher gate density, while a darker shade of green indicates low gate density.

Following the aforementioned consideration, in order to extract the neighbor gates of a given gate for being used in the toolchain, there exist two different approaches:

- The exhaustive method is based on elaborating the list of all gates. Each gate is analyzed by comparing its Euclidean distance with its layout physical position.

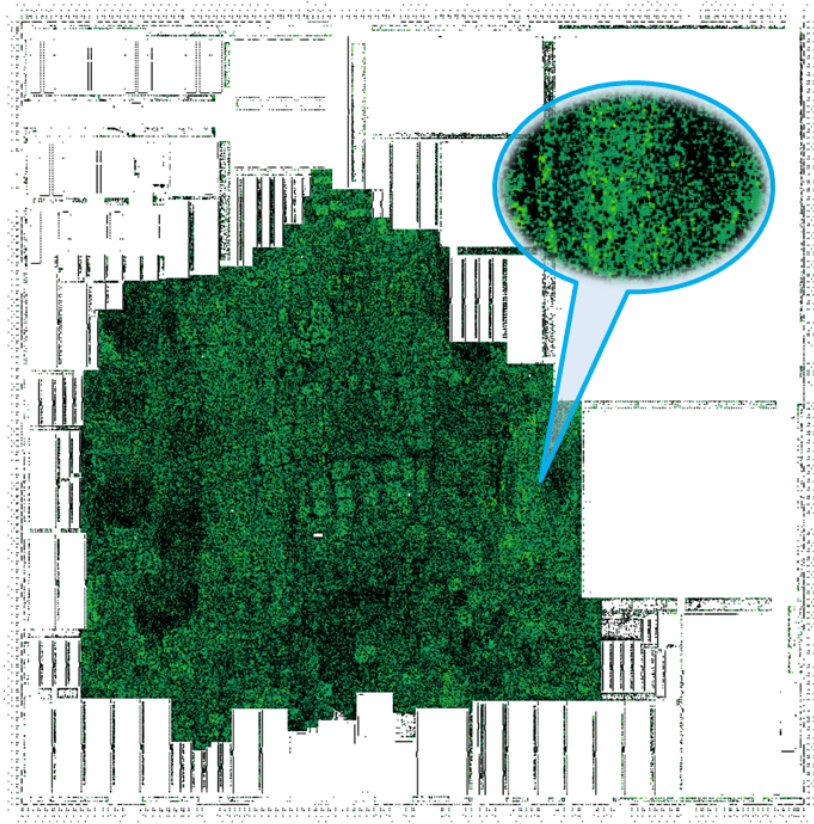


Fig. 2.12 Density-colored heatmap for the DUT

- The heuristic method is based on the DBSCAN algorithm to extract the neighbors for a given logic gate by using as internal metric the Euclidian distance with a fixed internode distance of $6\mu m$.

Table 2.15 compares the execution time of the Exhaustive method and the heuristic one.

Table 2.15 Comparing the execution time of the exhaustive and the heuristic approach.

Analysis approach	Execution Time
Exhaustive	20 days
Heuristic (DBSCAN)	654.13 s

As seen from Table 2.15, the execution time of the exhaustive method is 1000 orders of magnitude more than the heuristic one. This substantial reduction in the execution time allows for analyzing more complex DUT.

Regarding the exhaustive method, the runtime memory consumption is stable, and it is less than 4 GBytes due to the elaboration of a single gate at the time. An important aspect to mention, rather than the execution time of the heuristic method is the runtime memory consumption as Figure 2.13 depicts. Due to the nature of

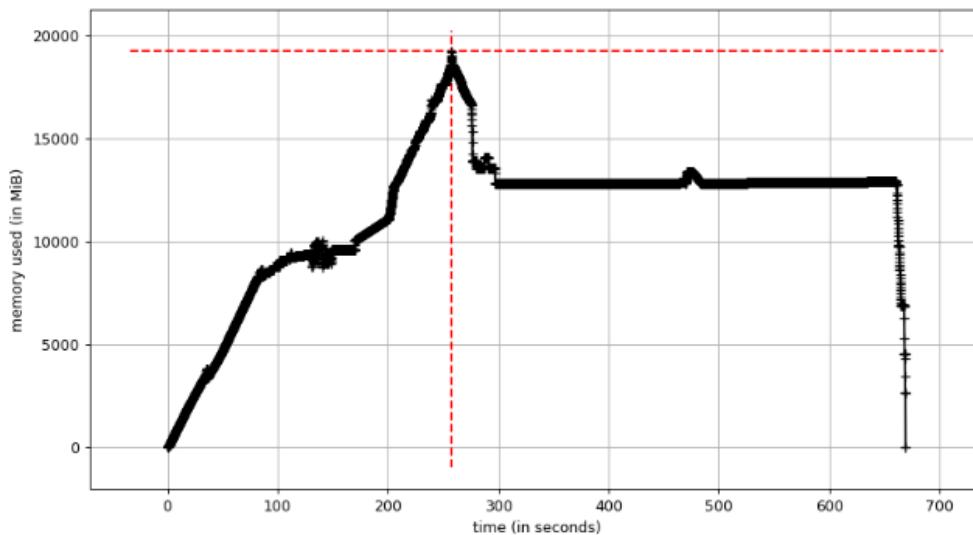


Fig. 2.13 Runtime memory consumption of the heuristic method.

the heuristic method, it can analyze more than one gate in parallel. Therefore, its memory consumption is unstable, as seen from Figure 2.13. The initial ramp-up is

due to the file reading where the physical positions of gates are stored. On the other hand, the peak in memory consumption is when the DBSCAN starts its computation. It creates for each gate a cluster, thus the peak in the memory usage, and it gradually merges neighborhood clusters until a stable configuration.

Whereas a heuristic method is used, it dramatically relieves the execution time, but it impacts the accuracy of the analysis compared to the exhaustive method. Therefore, it is fair to report the accuracy of the analysis by utilizing a confusion matrix shown in Table 2.16.

Exhaustive	Heuristic	Close	Far
Close		30 M	1 M
far		75M/8 M	200 K
Far		75 M	200 K

Table 2.16 Confusion matrix showing the accuracy of the heuristic method compared to the exhaustive.

It can be observed that the heuristic method returns a larger set of close nodes, i.e., about 30 million pairs. On the contrary, the heuristic method returns 75 million close pairs and misses almost 1 million far pairs. Overall, the exhaustive and the heuristic methods discard coherently 200 thousands of pairs. Therefore, the accuracy of the heuristic method is 95,9%, which is acceptable for the other analysis.

Metrics Weighting

Downstream the topology and VCD analyses, the single or multi-point metric can be enhanced with layout awareness capabilities. As for the evaluation of the stress metrics, experiments have been performed to show how layout awareness affects the metrics and their computational costs regarding time and memory consumption.

Table 2.17 details results starting from the single point metric, the toggle activity, which is enhanced with layout awareness coverage, i.e., weighting the metric based on the density surrounding a given logic gate.

As seen from Table 2.17, the required execution time and runtime memory usage remains constant across different stress patterns. The reason behind the constant values of memory and execution time is the nature of the input file containing the list of signals and their toggle activity. Those files have the same number of signals.

Stress pattern	Toggle coverage	Layout-aware coverage	Execution time	Rt Memory Usage
32ATPG	82.49 %	82.5 %	473.69 s	31 GB
12ATPG+	83.21 %	83.14 %	492.75 s	31 GB
1024PR	91.53 %	90.32 %	508.43 s	31 GB
LBIST	85.49 %	87.5 %	478.88 s	31 GB
MBIST	10.08 %	7.9 %	510.60 s	31 GB
FUNCT	10.59 %	8.56 %	612.38 s	31 GB

Table 2.17 Single point stress metric coverage.

Thus the same size and the script introducing layout awareness in the simple stress metric is independent of the stress pattern.

On the other hand, Table 2.18 reports the results on the multiple-point metric enhanced with the layout awareness.

Stress pattern	Neighbors coverage	Layout-aware coverage	Execution time	Rt Memory Usage
32ATPG	80.47 %	80.48 %	328.42 s	15 GB
12ATPG+	79.87 %	79.84 %	331.10 s	15 GB
1024PR	87.04 %	87.04 %	278.38 s	15 GB
LBIST	83.96 %	83.96 %	335.18 s	15 GB
MBIST	3.11 %	2.78 %	289.64 s	15 GB
FUNCT	28.58 %	28.56 %	286.61 s	15 GB

Table 2.18 Multiple point stress metric coverage.

Table 2.18 confirms the same results seen in Table 2.17, i.e., the execution time and the runtime memory usage are constant across different stress patterns, and they only depend on the input file size and its nature.

Furthermore, in order to prove the effectiveness of layout awareness metrics, Figure 2.14 visually represents how the weighted and unweighted activity metrics evolve concerning the number of applied structural patterns in the OpenRISC 1200 [35].

When just a few patterns are used, the layout-aware metric is lower than the unaware one, whereas when more than eight patterns are applied, the layout-aware metric tends to have higher values. Consequently, a much more significant part of the denser areas of the DUT is being covered.

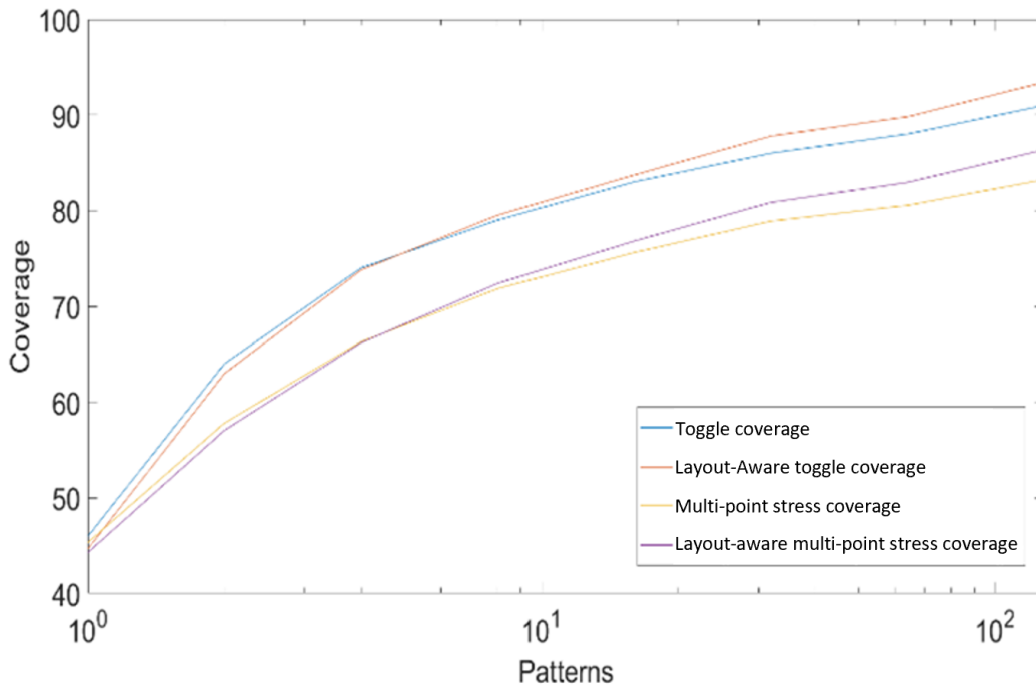


Fig. 2.14 Evolution of BI metrics.

This kind of behavior captures the way the patterns stimulate the different parts of the DUT. Indeed, with just a few patterns applied, the stimulation is nicely “spread” across the DUT. On the contrary, when many patterns are applied, the activity concentrates on the denser parts of the DUT. In this way, the contribution to the layout-aware metric tends to increase. Eventually, the layout-aware metric values exceed those provided by the unaware ones.

2.3.11 The Set Tool

The time and memory costs required by the Set tool mainly depend on the number of signals analyzed and the number of input files compared and merged. File reading is the slowest step of this stage, whereas analyzing the sets require fewer resources than the file reading step.

Table 2.19 shows the memory and time consumption trend over an increasing number of files of a fixed size of 2.3GB, performing all the available operations on the tool, which includes comparing and merging capabilities.

Number of Input files	Execution Time	Average Memory Usage
4	459 s	26.1 GB
6	500 s	37.1 GB
8	544 s	48.2 GB
10	676 s	59.3 GB

Table 2.19 Profiled execution of the Set Covering tool.

Memory increases proportionally with the number of files involved. However, the execution time increases only by a small amount. As more and more files get added, profiling shows that moving them to the proper data structure is the most costly operation, with the operation of saving the results. On the other hand, the output file size is always the same as the input file.

2.3.12 The Hierarchical Analysis Tool

In order to analyze the stress pattern and its strength in terms of coverage in the entities and sub-entities composing the DUT, a hierarchical analysis is performed.

Table 2.20 presents a profiled execution of the tool for a different number of entities for which the coverage is extracted. Moreover, runtime memory usage and execution time for different execution is presented, as well as the size of the output file.

Extracted entities	Output File Size	Execution Time	Rt Memory usage
2	172 MB	1,140 s	36.7 GB
4	172 MB	1,119 s	36.7 GB
6	172 MB	1,245 s	36.7 GB
8	172 MB	1,124 s	36.7 GB

Table 2.20 Profiled execution of Hierarchical analysis tool.

Table 2.20 depicts a constant execution time and memory consumption independently from the stress pattern. Therefore, its execution time, output file size, and memory usage is directly proportional to the number of signals in the input file, hence, in the DUT.

The high memory consumption is due to the internal data structure of the tool holding all the information for each entity within the DUT.

2.3.13 The Chip-Surface Stress Plotter

A plotter tool is used for troubleshooting, visually, the eventual coverage loss of a stress pattern or showing weaknesses due to the superimposition of different stress patterns. A stress-colored heatmap is produced by exploiting the physical placement of logic gates in the DUT layout.

Table 2.21 shows the runtime memory usage and execution time and how they are affected by the resolution of the image. Regarding the input files, they are most of the same size (around 2.3 GBytes for the analyzed stress pattern and 2 GBytes for the file containing the physical placement of gates as coordinates).

Most of the time cost to build images is the file reading, which, again, it strongly depends on the number of signals and the number of gates in the DUT.

Up-scaling or down-scaling the image between resolution values of 1,000x1,000 to 10,000x10,000 does not impact performance as much as the file reading, due to the intrinsic parallelism of the drawing operation, despite the image generation being performed on the CPU rather than on GPU.

For example, Figure 2.17 contains output images representing the stress coverage from different stress patterns. In addition, Figure 2.17 highlights unstressed and stressed regions for different patterns. Figure 2.17a depicts a non-stressed region better strained by the pattern in Figure 2.17b. Similarly, the pattern in Figure 2.17d focuses on a module not well stressed by the pattern represented in Figure 2.17c. On the other hand, the same pattern does not activate enough memory ports and some functional units, i.e., the upper and lower left zooms in Figure 2.17d. Finally, Figure 2.17e and Figure 2.17f show how to adequately stress memory ports and functional units, respectively.

2.3.14 Wrapping up the flow

The BI for safety-critical devices includes different patterns, ranging from structural to functional. Therefore, as represented in Table 2.11, single patterns are superim-

Resolution	Average File Size	Execution Time	Rt Memory Usage
1000x1000 pixels	4 MB	535 s	3.9 GB
2000x2000 pixels	16 MB	537 s	3.9 GB
4000x4000 pixels	62 MB	559 s	3.8 GB
8000x8000 pixels	245 MB	532 s	4.8 GB

Table 2.21 Profiled execution of Chip-surface stress plotter.

posed and analyzed to understand their weaknesses and ability. This activity allows the generation of the stress-colored heatmap plot for the overall BI phase displayed in Figure 2.15. From these images, it is straightforward to distinguish the zones where the stress level is adequate from those needing additional patterns (such as the area inside the dotted red circle).

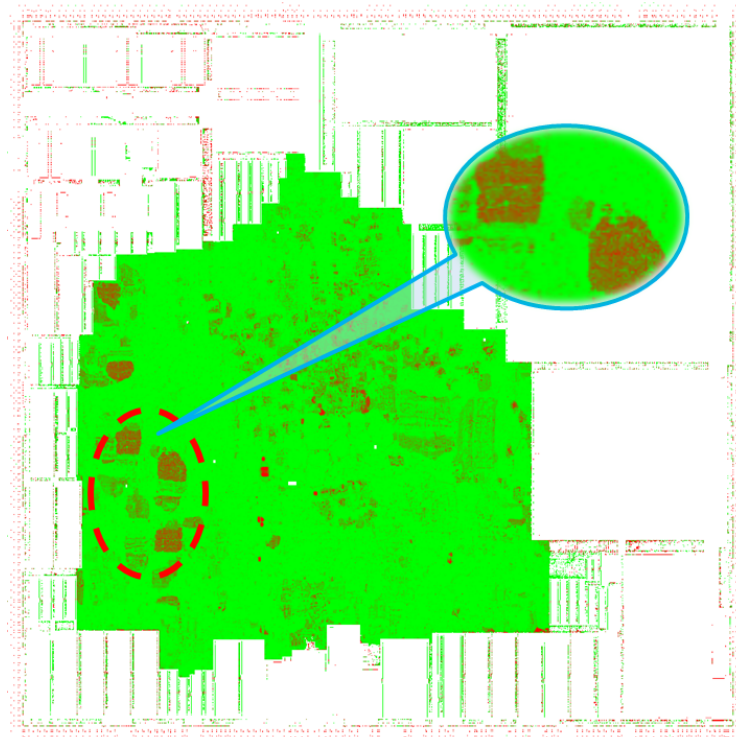


Fig. 2.15 Visualization of the overall stress provided by the superimposition of all stress patterns.

As pinpointed from Figure 2.15 and confirmed from the hierarchical analysis tool (Tool E), an unstressed zone exists that does not reach an acceptable level of

stress coverage. Therefore, we develop an additional functional stress pattern that can provide additional stress as represented in Figure 2.16.

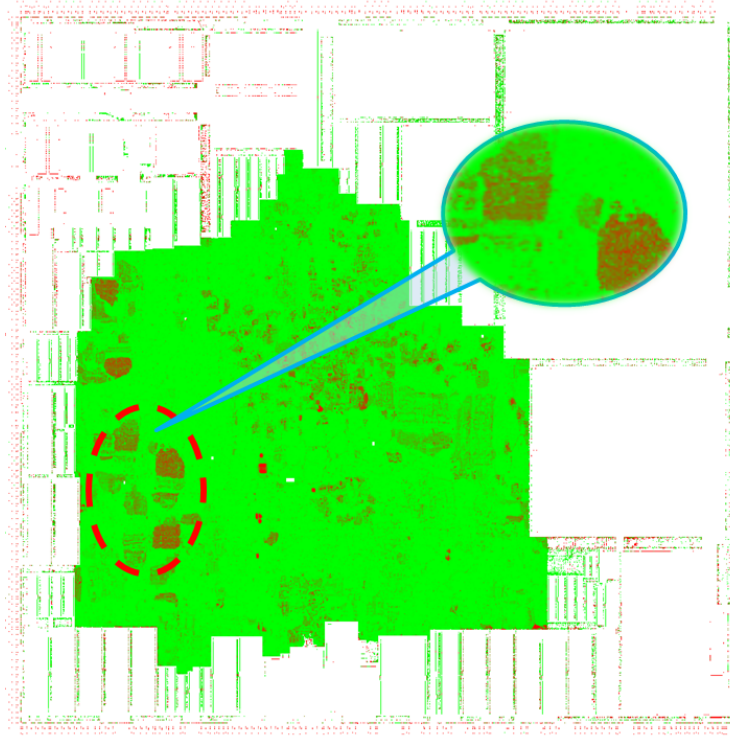


Fig. 2.16 Visualization of the overall stress provided by the superimposition of all stress patterns, plus the additional functional stress pattern targeting the identified unstressed module

As shown from Figure 2.16, the ad-hoc developed functional pattern stresses the region of interest, effectively increasing the stress coverage.

2.3.15 Conclusions

With this toolchain, we are able to perform a testing flow starting from the logic simulation, up to the visualization of its results. Moreover, we are able to merge more simulations, gathering an effective insight on multiple tests performed subsequently, while being able to save disk space by removing old VCD files.

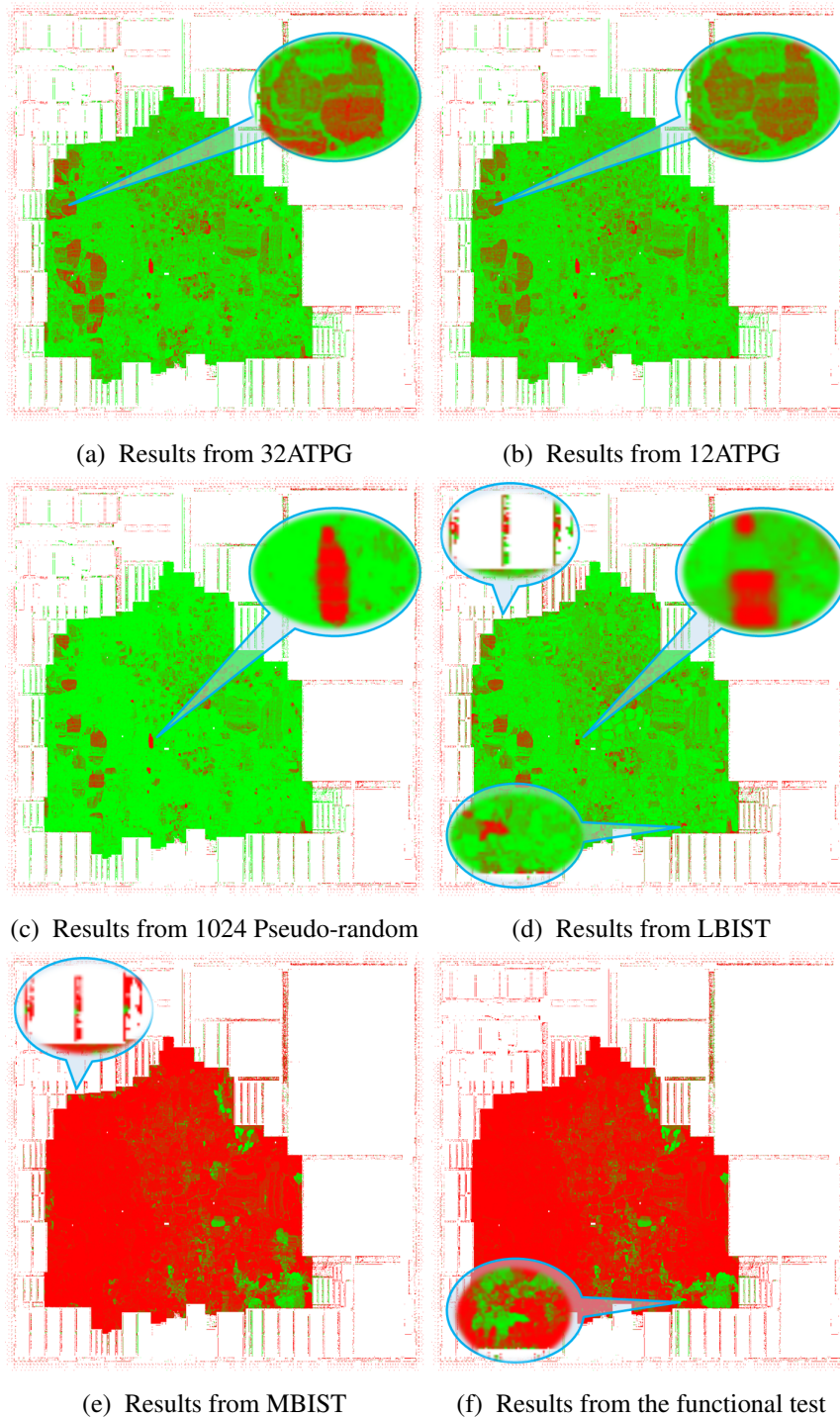


Fig. 2.17 Stress-colored heatmaps in terms of toggle coverage with detailed zoom on some regions.

2.4 Connectivity: A new metric

In this section, I will present the new metric we developed, called Connectivity. This section will talk about the work developed and published in [19], with some details and future work that will be presented.

2.4.1 The Proposed Methodology

The methodologies analyzed in Section 2.3, such as SBST, TDBI, and SLT, have extremely long evaluation times. Logic and fault simulation are the most expensive phase of creating high-quality functional test programs. Therefore, reducing the number of fault simulations is very important in the testing area. Our methodology addresses these aspects.

Two of the most adopted methodologies in the security research domain are “symbolic execution” and “dynamic taint analysis” [56–59]. Both these strategies analyze some software piece of code and grade it from the security perspective. Symbolic execution automatically builds a logical formula describing a program execution path, and it reduces the problem of reasoning about a program to a logic domain. Dynamic taint analysis runs a program and observes which computations are affected by predefined taint sources, such as the user inputs.

Following the dynamic taint analysis paradigm, we propose a technique to perform a preliminary evaluation of a functional test program without the necessity of a logic or fault simulation step in the early development phases. The flow of our methodology is illustrated in Figure 2.18, which major product is the computation of a novel metric called “connectivity”. The connectivity value is fast to compute and can effectively guide the development, especially during the early stages.

Previously designed programs in C language or Assembly code [60, 61] are used as benchmark tests (top-left corner of Figure 2.18). These test programs are compiled and executed on the silicon device and the instruction traces are generated by adopting advanced debugging tools. Each trace contains the list of executed instructions, with all selected branches, all loops properly unrolled, and all actual registers and memory values generated by the chip during the execution of the machine code. Test programs are developed to exercise functional blocks of the device and their characteristics have a direct relationship with the verified functionalities and, as a consequence, with their

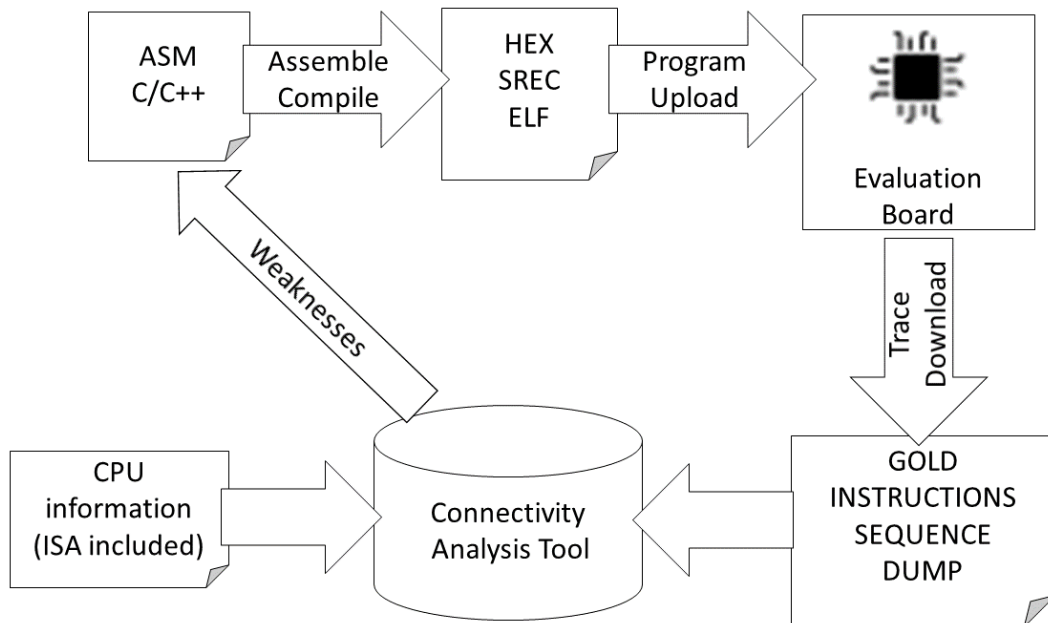


Fig. 2.18 The logic flow of our testing framework

final coverage. We consider their traces as gold functional executions (bottom-right block of Figure 2.18). Once they are produced, each trace is parsed and transformed into a Control and Data Flow Graph (CDFG) by the connectivity tool (bottom-center block of Figure 2.18). Portability between different architectures and ISA is achieved by the CPU information file (bottom-left block). An important aspect to highlight is the independence of the proposed approach from single and multi-core executions of the functional test programs. The connectivity analysis is performed from a CPU register perspective, meaning that we investigate how the data propagates within CPU registers and across instructions of test programs. Therefore, even with a multi-core execution, the functional routines can be divided and analyzed based on the CPU running them. Moreover, a multi-core program shares common resources and data, and consequently, it needs some synchronization mechanism and atomic operations in order to avoid non-deterministic values in memory which are taken into account by the CPU information file. In the connectivity step, every instruction is represented by a vertex and the information flow is represented by directed edges. We focus on writing and reading operations and their temporal dependency, independently from the sequence of operations and the program complexity in terms of lines of code. Thus, given a data value d in the trace (either a register or a memory location), we introduce two types of edges:

- *WAW* (Write-After-Write) edges represent two write operations on d back-to-back with no reading instructions in between.
- *RAW* (Read-After-Write) edges represent a write operation followed directly by a read operation on d .

Once it has been built, the CDFG is analyzed to extract the “connectivity” metric. The connectivity measures potential code weaknesses and guides test engineers to grade the quality of the functional procedures, rectify their problems, and insert proper signature checks to improve their quality. The main advantage of this procedure is that the entire process is extremely fast (up to seconds) when compared to a standard logic and fault simulation phase (requiring up to hours or days). Our analysis relies on the following observation.

Write-after-Write (*WAW*) instruction sequences occurring during the instruction flow over a shared addressable location causes the previously computed values to be overwritten and most likely lead to a loss of fault coverage. Conversely, Read-after-Write (*RAW*) instruction sequences propagate values along with the execution flow, and they can reach an observation point, possibly leading to an increase in fault coverage.

The following example illustrates the main concepts of our process.

example Let us consider the code segment reported in Table 2.22. The code includes three instructions belonging to PowerPC VLE Instruction Set Architecture and the operation they perform.

Cycle	Instruction	Operation
1	e_add2i r19, r18, 192	r19 = r18 + 192
2	subfme r6, r18	r6 = r6 - r18
3	e_add16i r6, r19, 35	r6 = r19 + 35

Table 2.22 An example of instruction sequence with the operation performed.

For the sake of simplicity, we may suppose to represent each instruction with a vertex node as shown in Figure 2.19a. The data written in r6 by instruction number 2 is immediately overwritten by instruction number 3. In our representation, this is represented by a *WAW* edge incident from vertex 2 and in vertex 3 (Figure 2.19b). Vice-versa, the write operation on register r19 performed by instruction one is

preserved by the reading operation of instruction 3 and this is represented by a RAW edge incident from node one and in node 3 (Figure 2.19b). Even if there are cases in which WAW edges may have a specific meaning, they somehow correspond to undesirable situations with information flow disruptions. These situations should never appear in ASM or compiled C/C++ functional programs. On the contrary, RAW edges indicate the standard operation flow and are beneficial.

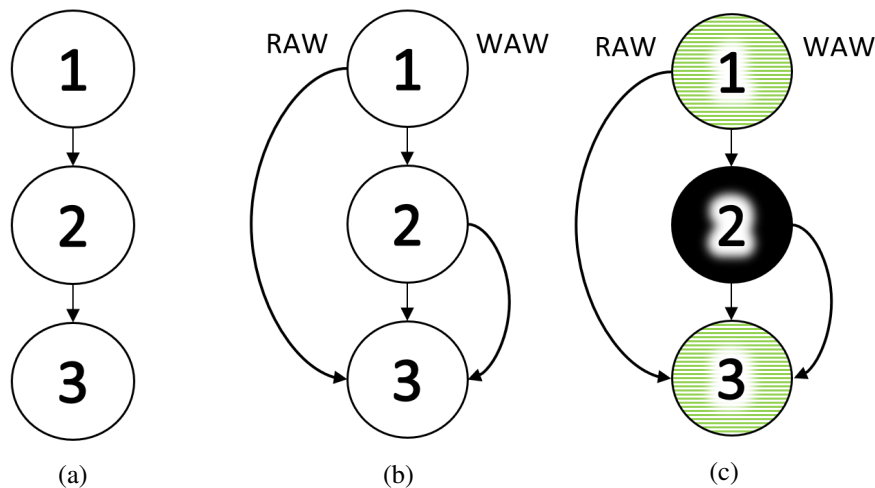


Fig. 2.19 A graphical representation of Example 2.4.1.

From a testing point of view, instructions 1 and 3 have may provide a positive effect of the fault coverage when the destination registers is propagated to an observable point. On the contrary, instruction 2 is not beneficial to fault coverage as the value of r6 is overwritten and cannot be propagated. As a consequence, instruction 2 is useless and can be canceled, or the program needs to be modified to unlock a possible propagation of r6. To sum up, WAW and RAW edges allow us to color the vertices of the graph as illustrated in Figure 2.19c and described in the following paragraphs.

The analysis we perform on the golden instruction dump verifies the presence of blocking situations similar to the one illustrated in the previous example. Analyzing the presence of WAW and RAW edges is essential, at least in the following contexts:

1. Randomization and evolutionary methods to automatically generate functional programs may naturally introduce WAW edges. Therefore, it would be beneficial to tune the generation of programs to minimize WAW edges and reach a high coverage faster.

2. Bugs may constantly be introduced in assembly software, i.e., even simple typos may introduce unwanted WAW sequences when writing low-level ASM code. Discovering an error after a fault simulation is frustrating, and a quick preliminary check would significantly reduce time loss.
3. It is often unfeasible to grade the fault coverage achieved by very long programs, such as benchmarks directly compiled in C/C++ language or OS boot procedures. In this case, checking potential weaknesses can be one of the few feasible measures in short time.

As shown in the following subsection, every instruction node of the CDFG is classified as blocked (black) or not blocked (green), depending on whether it propagates or not. We compute our *connectivity* metric as the fraction of non-blocked instructions over the total number of investigated instructions. A program with very high connectivity can be considered promising from the perspective of its potential fault coverage. Conversely, if the program shows a lot of blocked instructions and a low connectivity, it needs to be revised to avoid useless and time-expensive fault simulation stages. Furthermore, the connectivity measure locates the blocked instructions and may guide the designer to fix the proper components along the development process.

2.4.2 The Basic Algorithm

Although static and dynamic code analysis is performed in other fields [62, 56–59], in our approach, we process the code generated with a hardware debugger during the execution of functional programs on real silicon. Similarly to the taint analysis, our approach statically elaborates the instructions flow executed by the program to identify blocking situations within the golden instruction trace. Therefore, the proposed methodology recognizes critical edges in the CDFG, i.e., it identifies WAW edges preventing information to be forwarded along the program flow.

Table 2.23 reports a short snapshot of a golden instruction dump extracted from a manually developed test routine for the CPU adder unit. The first column indicates the order of execution; the second and third columns report the address and the mnemonic code of the relative instruction; columns SRC and DST specify the sources and destinations of each instruction.

Table 2.23 Instruction sequence, source, and destination operands.

Cycle	Address	Instruction	SRC	DST
1	0x0000_0000	subfme r19, r6	r19, r6	r19
2	0x0000_0004	e_add16i r6, r19, 35	r19	r6
3	0x0000_0008	e_add2i r6, r19, 192	r19	r6
4	0x0000_000C	e_add2i r6, r6, r1	r6, r1	r6
5	0x0000_0010	subfme r19, r6	r19, r6	r19
6	0x0000_0014	subfze r6, r3	r6, r3	r6
7	0x0000_0018	e_add16i r6, r19, 35	r19	r6

Based on this information, the basic algorithm runs through the steps illustrated in Figure 2.20:

1. In the first step, the one represented by Figure 2.20a, we build a CDFG in which each instruction is mapped onto a vertex and the data flow is represented through edges. We use two types of edges, namely RAW (reported on the left-hand side of the graph) and WAW (reported on the right-hand side of the graph).
2. During the second stage, Figure 2.20b, we perform a visit of the CDFG following WAW edges. Vertices are colored either in red or green color.
3. During the third and last step, Figure 2.20c, we perform a visit of the CDFG following RAW edges. Red vertices may become green or black.

To build the CDFG, we use the function reported in Algorithm 2. For each node (i.e., instruction) of the graph (line 1), we visit all subsequent nodes (instructions) of the graph (function Search) looking for the destination of the current node as source or destination of one instruction. WAW edges are inserted to connect “victim” instructions to “aggressor” instructions (line 6). Aggressor instructions overwrite the result of victim instructions. RAW edges are inserted to connect a predecessor and a successor instruction (line 8). Building the graph has a cost that is linear in the number of instructions of the gold model.

Function Build_Graph is followed by Algorithm 3 which visits the graph for the first time, considering only WAW edges. For each node of the graph G (line 1), we check the existence of an outgoing WAW edge: Each vertex with an outgoing

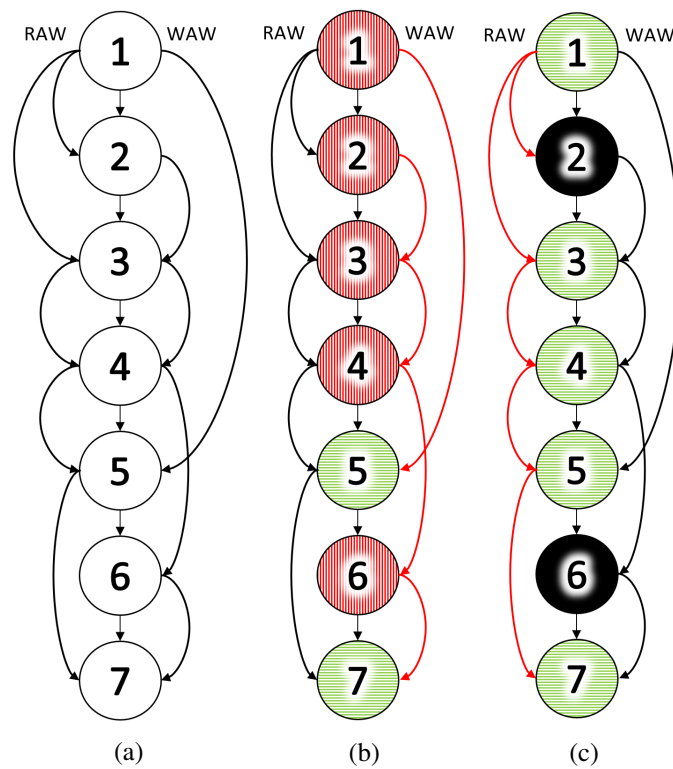


Fig. 2.20 The CFG of the code snippet of Table 2.23 is represented in Figure (a). Figure (b) illustrates the vertex-coloring process obtained following WAW edges, and Figure (c) the colors obtained at the end of the RAW visit.

Algorithm 2 Phase 1 of Figure 2.20a: From the gold trace to the CDFG. The procedure elaborates the execution dump and creates the graph with all WAW and RAW edges.

```

Build_Graph ()
1: for node in  $G$  do
2:   Search ( $G$ , node)
3: end for
Search ( $G$ , node)
4: for  $n$  in  $G > \text{node}$  do
5:   if  $\text{node}_{dst} = n_{src}$  then
6:      $\text{node}_{RAW} = n$ 
7:   else
8:     if  $\text{node}_{dst} = n_{dst}$  then
9:        $\text{node}_{WAW} = n$ 
10:      break
11:    end if
12:  end if
13: end for
14: return

```

WAW edged is colored in red (i.e., dark gray in the black-and-white reproduction of the picture) or in green (i.e., light-gray) if they do not present an outgoing WAW edge. For example, in Figure 2.20b instruction 4 is red (dark-gray) because it has an aggressor in instruction 6. On the contrary, instruction 5 is green (light-gray) because its computed value will reach the end of the program. Function WAW_Visit has a linear cost in the number of vertices of the graph.

Algorithm 3 Phase 2 of Figure 2.20b: The WAW visit.

```

Write_After_Write_Visit ( $G$ )
1: for node in  $G$  do
2:   if  $\text{node}_{WAW} \neq \emptyset$  then
3:      $\text{node}_{color} = \text{RED}$ 
4:   else
5:      $\text{node}_{color} = \text{GREEN}$ 
6:   end if
7: end for

```

The second visit essentially traverses RAW edges to finalize the color of all red nodes. The corresponding pseudo-code is reported in Algorithm 4. The structure of the code is similar to the one of the function Build_Graph in Algorithm 2. During the

visit, a red node can turn green (line 8) if there is a RAW edge connecting it ahead to a green vertex (line 7). On the contrary, the block suspect is confirmed, and the red color is updated to black (line 12). For example, in Figure 2.20b, instruction 4 is a case of a red node evolving into a green one, whereas instruction 6 is confirmed as a blocking vertex, and it becomes black. Function RAW_Visit, as all the previous procedures, has a linear cost in the number of vertices of our CDFG.

Algorithm 4 Phase 3 of Figure 2.20c: The RAW visit.

Visit (G)

```

1: for node in  $G$  do
2:   RAW_Visit (node)
3: end for

```

RAW_Visit (node)

```

4: if (nodecolor = RED) then
5:   for RAW_destination in RAW_edges of node do
6:     ret = RAW_Visit (RAW_destination)
7:     if ret = GREEN then
8:       nodecolor = GREEN
9:       return GREEN
10:    end if
11:   end for
12:   nodecolor = BLACK
13: end if
14: return nodecolor

```

Once all graph nodes are colored, we evaluate the connectivity of the code. The connectivity value is computed as the percentage of green (light-gray) nodes over all nodes in the CDFG, and it indicates the percentage of the instructions that are beneficial in terms of fault coverage. For example, in Figure 2.20, as 5 out of 7 instructions are finally green, the connectivity is 71.42%. The remaining two black vertices bring no valuable information to the assertion/check part of the code. Indeed, instructions 2 and 6 will never contribute to the fault coverage since their results are overwritten before propagating elsewhere.

2.4.3 Optimized Algorithm

The algorithm previously described can be optimized by removing the WAW edges and the WAW visit. The method illustrated in Section 2.4.2 creates WAW arcs to

find problematic instructions. However, WAW arcs are not necessary to obtain the final result, and we can perform the WAW visit while we build the CDFG, reducing the computation cost and improving the memory efficiency.

More specifically, when building the CDFG, instead of linking two nodes with a WAW edge, we evaluate an initial coloring of the node from which the edge leaves. To detect this preliminary color we check whether the node which would be overwritten also contains a RAW edge. In the positive case we set its color to red, otherwise we set it to black. For example, in Figure 2.20, instruction 6 could be set to black already during the construction of the CDFG as the destination is overwritten, and none of the following nodes has a RAW dependency from it.

The optimized procedure building the CDFG, i.e., Algorithms 2, is reported in Algorithm 5.

Algorithm 5 The optimized algorithm for phase 1 (originally, Algorithm 2). Several nodes are colored up-front, saving subsequent computation time.

Build_Graph (G)

```

1: for node in  $G$  do
2:   Search ( $G$ , node)
3: end for

```

Search (G , node)

```

4: for nextNode in  $G >$  node do
5:   if nextNodesrc = nodedst then
6:     nodeRAW = nextNode
7:   end if
8:   if nextNodedst = nodedst then
9:     if nodeRAW  $\neq \emptyset$  then
10:      nodecolor = RED
11:    else
12:      nodecolor = BLACK
13:    end if
14:    break
15:  else
16:    nodecolor = GREEN
17:  end if
18: end for

```

2.4.4 Load/Store Instructions

When we find a load or a store instruction in the code, we have to modify the previous analysis to take into consideration memory locations. Memory values are manipulated as virtual registers. The golden instruction sequence is first enriched with information related to registers to locate memory locations. Then, the current values of these registers is used to compute the virtual register address of the memory location. The following example clarifies this strategy.

example The code snippet reported in Table 2.24 includes load and store instructions from and to memory locations in addition to arithmetic operations. For the sake of simplicity, all memory locations have a size of 1 byte.

Table 2.24 Instruction sequence, source and destination operands.

Cycle	Instruction	SRC	DST
1	e_stb r0, 0(r1)	r0, r1	mem(0+r1)
2	e_add16i r2, r0, 35	r0	r2
3	e_lbz r2, 0(r1)	mem(0+r1), r1	r2
4	e_add2i r2, r1, r2	r1, r2	r2
5	e_stb r2, 0(r1)	r2, r1	mem(0+r1)
6	subfze r1, r3	r1, r3	r1
7	e_add16i r1, r0, 35	r0	r1

Figure 2.21a reports the corresponding graph and Figure 2.21b the final coloring scheme returned by the RAW visit. The example shows as we handle memory addresses as registers. For example, instruction one stores a value in the location pointed by register r1. A load instruction accesses the same location at position 3. Therefore, instruction one is finally labeled as green (light-gray). Moreover, instruction 5 is marked as green because the value written in memory is propagated until the end.

2.4.5 Branch Instructions

Another extension to our initial algorithm is required to manage branch instructions. In order to analyze and classify branches as green or black, we modified our algorithm in the following directions: dified the algorithm in the

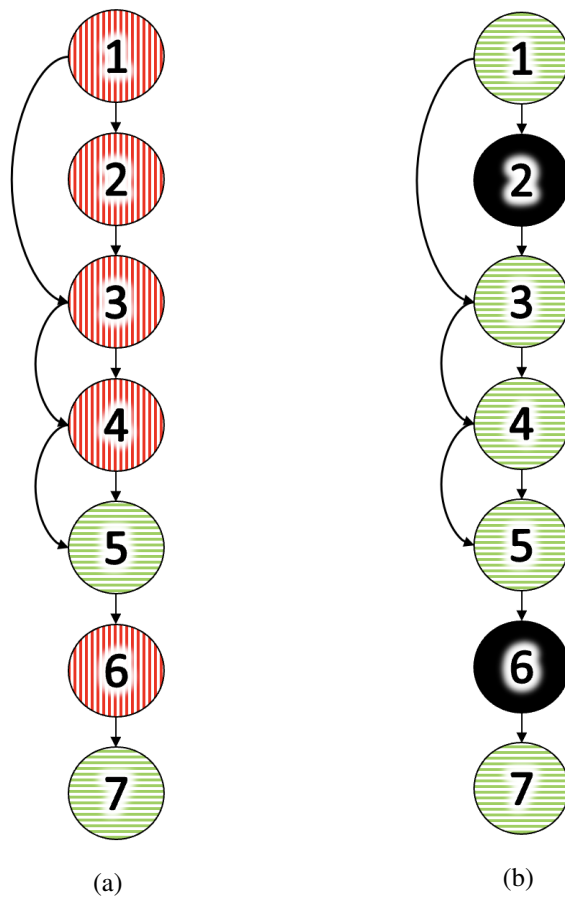


Fig. 2.21 An example including arithmetic operations and load/store instructions.

- During the regular visit, conditional branches are left undecided and temporarily colored in orange.
- An additional visit is performed to find the alternative branch destination among the instructions executed after the conditional statement.

If the branch's alternative destination address cannot be found, then, the branch is colored in black.

Conversely, when the branch alternative destination is in the traced instruction flow, the visit checks whether the incorrect branch execution leads to a different signature or not and color the branch in black or green.

However, notice that just some cases can be resolved by this strategy as our algorithm can color the branches in green or black only when also the alternative branch can be found in the code.

example The code reported in Table 2.25 includes a branch generated by a simple if-then-else construct.

Table 2.25 Instruction sequence, source and destination operands.

Cycle	Address	Instruction	SRC	DST
1	0x0000_0000	e_li r0, 0		r0
2	0x0000_0004	cmpl r0, 1	r0	cr
3	0x0000_0008	e_beq jump	cr	
4	0x0000_000C	cmpl r0, 0	r0	cr
5	0x0000_0010	e_add16i r1, r2, r3	r2, r3	r1
6	0x0000_0014	jump: e_add16i r1, r0, r1	r0, r1	r1

Figure 2.22a reports the result of a first visit, leaving the branch node as undecided. The second visit, whose result is represented in Figure 2.22b, evaluates the signature under the hypothesis of taking the wrong branch path. In this specific case, the instruction that would be reached if the branch decision was wrong is included within the instructions that follow the branch itself. A new orange-colored edge is added to the figure to highlight this situation, and it appears that some instructions would not be executed if the branch was wrongly executed (taken if it was not taken, and vice-versa).

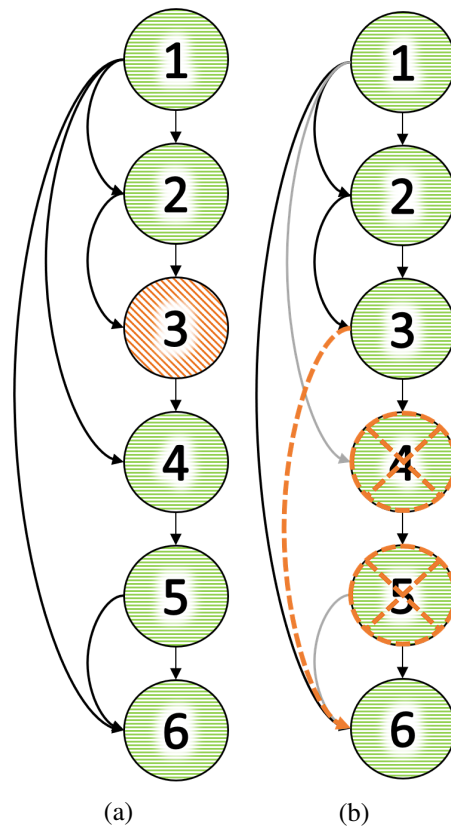


Fig. 2.22 A graph example including a branch instruction.

The instructions that may be skipped are green. Therefore, the value of the final register may change, and the signature value may potentially be compromised. Indeed, the branch node is labeled with a green color. Conversely, if all skipped instructions had been labeled as black nodes, they would not contribute to the signature value. In this case, the branch would be colored black, as illustrated in Example 2.4.5.

example 2 Let us focus on the code reported in Table 2.26. Following Example 2.4.5, our first visit leaves the branch node undecided. The second visit determines that the signature would not change under the hypothesis of taking the wrong branch path, i.e., all skipped instructions are labeled as black nodes. As a consequence, the branch is colored black. The outcomes of the two visits are illustrated in Figure 2.23.

Table 2.26 Instruction sequence, source and destination operands.

Cycle	Address	Instruction	SRC	DST
1	0x0000_0000	e_li r0, 0		r0
2	0x0000_0004	cmpl r0, 1	r0	cr
3	0x0000_0008	e_beq jump	cr	
4	0x0000_000C	e_add16i r1, r2, 1	r2	r1
5	0x0000_0010	e_add16i r1, r0, 1	r0	r1
6	0x0000_0014	jump: e_add16i r1, r0, r2	r0, r2	r1

2.4.6 Multiple Destination Instructions

To complete the functionalities of the proposed method, we need to consider one last extension, as the code can include instructions with multiple destinations. In this case, we extend our analysis to target destinations instead of instructions as illustrated in Example 2.4.6.

example Table 2.27 reports a code snippet where several instructions have more than one operand destination.

For the sake of completeness, Figure 2.24 shows our initial graph (Figure 2.24a), the one with WAW and RAW edges (Figure 2.24b, introduced in Section 2.4.2), and

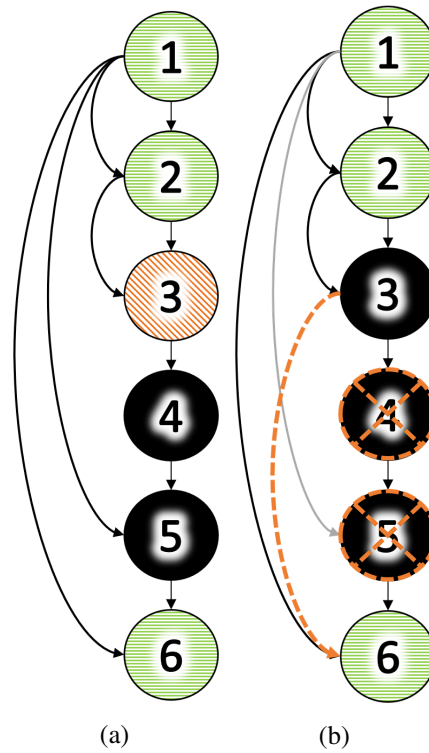


Fig. 2.23 Another branch instructions example.

Table 2.27 Instruction sequence with multiple destinations.

Cycle	Address	Instruction	SRC	DST
1	0x0000_0000	subf r0, r1	r0, r1	cr, r0
2	0x0000_0004	e_add16i r1, r0, 35	r0	r1
3	0x0000_0008	e_add2i r1, r0, 192	r0	cr, r1
4	0x0000_000C	e_add2i r1, r1, r1	r1	cr, r1
5	0x0000_0010	subf r0, r1	r0, r1	cr, r0
6	0x0000_0014	subf r1, r2	r1, r2	r1
7	0x0000_0018	e_add16i r1, r0, 35	r0	r1

the one obtained after the final coloring phase (Figure 2.24c). Coloring is performed considering “single” destinations. Consequently, each graph vertex includes a color for each of the destination fields considered by the instruction.

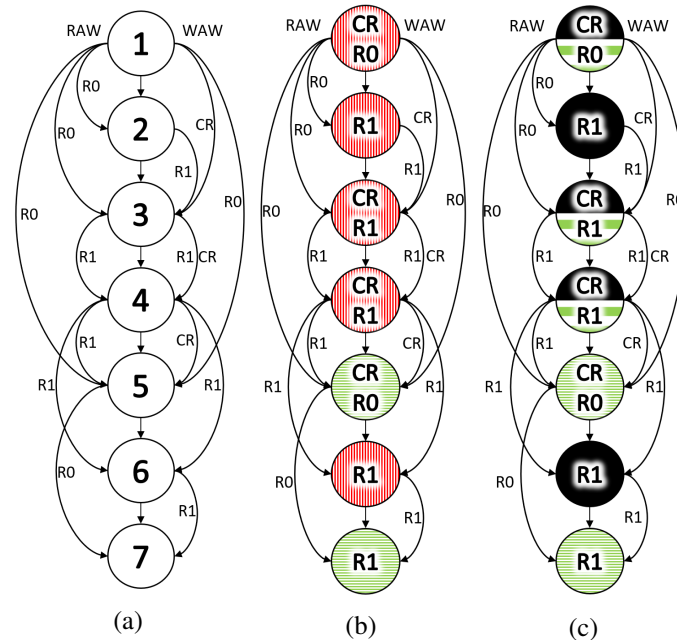


Fig. 2.24 A code snippet with instructions with multiple destinations: Our instruction-oriented analysis is modified to be destination-based.

Algorithms 6 and 7 report the modified functions to build and visit the CDFG in case of multiple destinations. The data structure of a node has been changed to manage multiple destinations. Moreover, if a node has at least one green destination, i.e., the instruction propagates at least some partial result, we consider the data as propagated. When we consider multiple destinations, the metric connectivity needs to be computed in a slightly different way. We first assign a connectivity percentage to every node. Then, we evaluate the overall connectivity value as the average of the connectivity of all nodes.

For instance, in Example 2.4.6, node 1, 3 and 4 are 50% connected, instruction 5 is 100% connected, and instructions 2 and 5 are not connected at all. Therefore, the overall connectivity metric value is computed as $(3 \cdot 50\% + 2 \cdot 100\%) / 7 = 50\%$.

Algorithm 6 Build graph function that elaborates the execution dump considering multiple destinations.

```

Build_Graph ()
1: for node in G do
2:   for dst in node do
3:     Search (G, node, dst)
4:   end for
5: end for
Search (G, node, dst)
6: for n in G > node do
7:   if nsrc = dst then
8:     nsrcRAW = dst
9:   end if
10:  if ndst = dst then
11:    if dst is never read then
12:      dstcolor = BLACK
13:    else
14:      if dst is read then
15:        dstcolor = RED
16:      end if
17:    end if
18:    return
19:  end if
20: end for
21: if dst never found then
22:   dstcolor = GREEN
23: end if

```

Algorithm 7 RAW visit considering multiple destinations.

```

Visit (G)
1: for node ∈ G do
2:   RAW_visit (node)
3: end for
RAW_visit(node)
4: for dst in node do
5:   if dstcolor = RED then
6:     for RAW edge i do
7:       if dstcolor ≠ GREEN then
8:         dstcolor = RAW_visit (RAW edge[i])
9:       end if
10:    end for
11:  end if
12: end for
13: if Any of dst in node = GREEN then
14:   return GREEN
15: end if
16: return BLACK

```

Program name	Code size [bytes]	Execution Time [cc]	Executed instructions	Connectivity [%]	Grading Time [s]	Stuck-at fault [#]	Stuck-at fault coverage [%]	Fault simulation CPU time [h]	
Adder	2,708	3,388	1,037	91.38	40	19,760	92.56	24.39	
Multiplier	original	5,612	7,002	1,256	68.96	33	64,004	92.3	46.09
	improved	5,584	6,981	1,249	72.03	33	64,004	92.3	43.14
Floating Point	13,948	78,286	17,644	96.4	680	70,300	90.78	1,407.11	
Shifter	original	6,344	7,226	2,080	92.38	80	17,128	86.19	169.61
	improved	6,344	7,226	2,080	92.63	80	17,128	89.48	39.86
Count-zeros	1,408	3,823	1,112	94.49	43	3,096	86.81	12.19	
Bit-wise Logical	680	513	146	97.63	5	2,828	95.00	5.65	
Load-Store	3,016	4,228	1,051	53.28	40	12,865	52.54	11.19	
Branch Target Buffer	4,476	31,135	3,462	68.45	133	19,990	71.16	66.86	
Random	1M	4M	4M	1M	38.5	NA	NA	NA	NA
	10M	40M	40M	10M	76.0	NA	NA	NA	NA
RTOS	original	130,024	210,816	28,736	75.61	1,108	1,528,461	NA	7,154.92 (est.)
	enhanced	131,906	467,840	34,998	76.91	1,349	1,528,461	NA	10,655.26 (est.)

Table 2.28 Test programs evaluation. All branches without an alternative are marked as black in our analysis. NA means the data is not available.

2.4.7 Experimental Results

The experimental evaluation embraces several testing scenarios sharing the same automotive device, i.e., a micro-controller of the SPC58 family, manufactured by STMicroelectronics. Section 2.4.8 details the characteristics of the chip. Section 2.4.9 reports our evaluation of SBST programs belonging to a Core Self-Test library [60]. Section 2.4.10 describes how to use the connectivity metric to rapidly generate stressful functional programs that also own fault coverage capabilities. Section 2.4.11 reports some experiments to evaluate SLT applications.

2.4.8 The Industrial Device under Analysis

Our test programs are executed on an automotive microprocessor belonging to the SPC58 family from STMicroelectronics. This micro-controller features multiple cores, many modules (such as timers), and several communication modules. The microprocessor is usually employed in safety-critical applications in modern vehicles. Overall, the unit includes about 20 million gates. Each one of its CPUs incorporate about 1.5 million stuck-at faults. Figure 2.25 illustrates the experimental setup, including the micro-controller and the hardware debugger, adopted to validate our methodology.

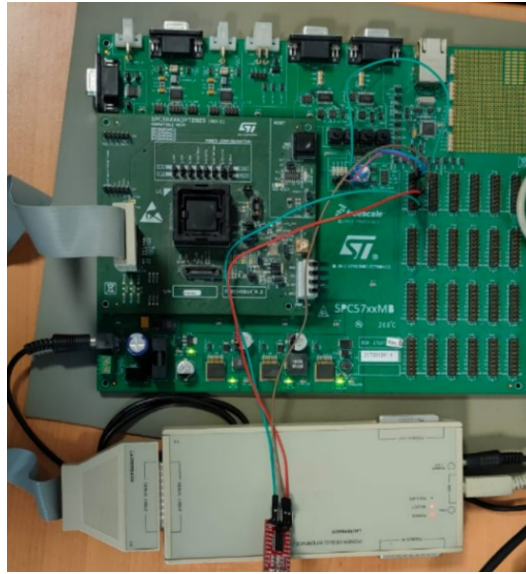


Fig. 2.25 The experimental setup including a development board with the SPC58 microcontroller and a hardware debugger.

2.4.9 Evaluation of SBST Programs

During the first set of experiments, we analyze SBST programs belonging to a Core Self Test library developed by skilled test engineers. SBST programs are used during standard working operations to target all potential stuck-at faults incorporated by several components within the CPU core.

The first section (the one on top) of Table 2.28 reports our findings in this area. In order to understand the complexity of each program, we include their size (in bytes), their execution time (in clock cycles), and the number of instructions executed. Columns 4 and 5 (i.e., columns “Connectivity” and “Grading Time”, respectively) indicate the connectivity (as defined in Section 2.4.1) and the running time of our methodology. The computation time is mainly due to the extraction of the instruction dump. The last three columns are dedicated to fault-related information and report the number of stuck-at faults, the fault coverage, and the single-threaded fault simulation time for each program.

The benefit of using the connectivity metric are manifold. The test programs that show high fault coverage, most frequently reflected in a high connectivity. Similarly, programs with a mediocre coverage also return a weak connectivity. Experimental results encourage to report that there is a correlation among fault coverage and

connectivity metric when evaluating mature SBST programs. Such a correlation could not be true at the very beginning, with short programs that can be well connected but poor in terms of testing abilities. It never happens that a program with high coverage shows low connectivity.

The implemented flow allowed us to improve some of the test programs. In particular, for the *Shifter* test program, we identified a lack of connectivity due to a single instruction affected by a WAW hazard. Once this issue was rectified, the connectivity increased from 92.38% (line “original”) to 92.63% (line “improved”). Similarly, the fault coverage increased by more than 3%, from 86.19% to 89.48%. For the *Multiplier* test program, a few instructions were marked as useless by our analysis, i.e., the code did not propagate the value they compute to an appropriate signature point. Therefore, we canceled these instructions. In this way, we reduced the execution time and the memory footprint of the program, and we maintained its original fault coverage (i.e., 92.3%) at the same time. The memory reduction was about 1%, and it was proportional to the number of executed instructions.

However, this program includes hand-written routines and randomly generated parts that may have introduced redundant instructions. Therefore, removing redundant instructions could speed up the program by maintaining the same fault coverage. This process leads to a smaller program able to test the multiplier faster, saving CPU time.

2.4.10 Evaluation of BI Programs

In this section, we describe how to use our methodology to improve the quality of functional programs created for the Test during BI (TDBI). To generate stressful functional programs, we use the Evolutionary Optimizer microGP [63] to maximize the switching activity of the cores. Such an optimizer is provided with a set of instructions and operands, and it evolves a set of candidate assembly programs from one generation to the next one. Each program is simulated and ranked to increase its switching activity progressively as the algorithm gives more weight to individuals with a high connectivity value. In fact, for TDBI we would also like to have a certain level of fault coverage. Indeed, a high connectivity value imply an increased the possibility of detecting errors during the execution of the stress program.

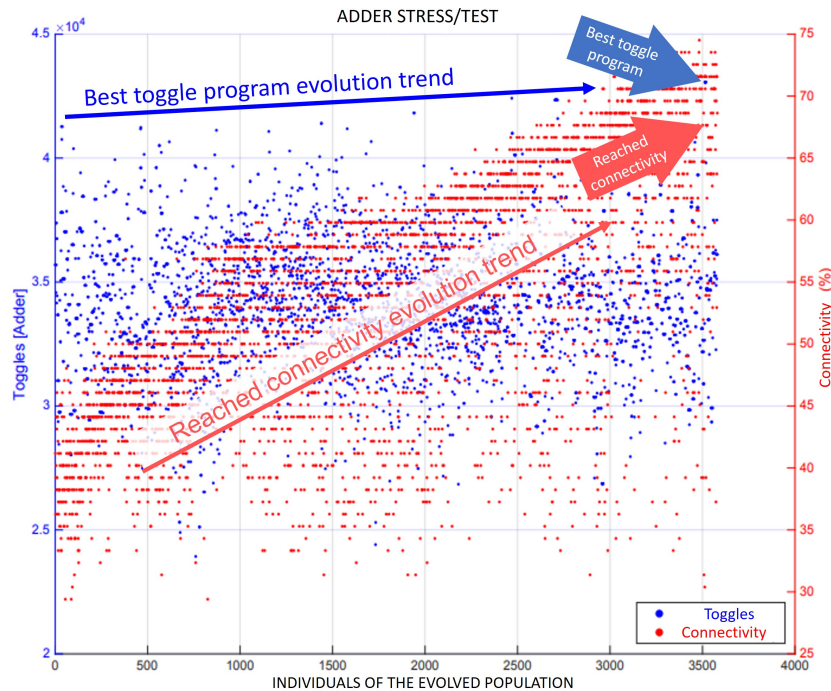


Fig. 2.26 Evolution of the population of ASM programs generated by the optimizer microGP.

Figure 2.26 illustrates the overall evolution of the generated stress programs addressing the adder unit of the chip. Every evaluated program is represented with two dots on the chart. Blue dots are related to the produced toggle activity, whereas red ones indicate the connectivity. The number of generated individuals along the evolutionary process is reported on the X-axis. The right-hand side Y-axis reports the average toggle activity per gate value achieved by the program on every gate of the adder. The left-hand side Y-axis reports the metric connectivity value. Evolution trends are characterized by a slow but constant increase of the toggle activity and fast growth from low to decent connectivity values. Arrows pinpoint the best individual, i.e., the most stimulating program, and its measured connectivity.

To measure the scalability of the methodology, we use randomly generated stress programs counting up to millions of instructions. The second section of Table 2.28 reports data on two very long stress programs. The first one is composed of one million sequential instructions with a limited set of registers used by each instruction. The second one is a random program with 10 million instructions, using only arithmetic instructions in which few registers are used to pass values along the logic flow. Overall, the analysis takes a few seconds for the 1M instruction

program and less than 5 minutes for the 10M instruction program. In the first case, the connectivity is quite low because of the large set of registers used, making it more difficult to have high intrinsic connectivity, as the random generator has more freedom for possible mistakes. The second program is more connected because of the smaller number of registers used.

2.4.11 Evaluation of SLT programs

The last section of Table 2.28 (the bottom section marked as “RTOS”) reports an analysis of an RTOS used for SLT programs. The device can run a basic and an advanced version of the Micrium-C OS III [64]. In this case, our analysis is motivated by the fact that the fault simulation of the original RTOS and, more generally, of extended test programs, is practically unfeasible [23], due to the enormous computation times (estimated to be around one year or more). Consequently, we apply the connectivity analysis to the original version of the RTOS. As the table shows, we obtain a connectivity value of 75.61%. We also analyze a different version of the RTOS, enhanced with a signature computation in every context switching between tasks. The signature in every context switching is used to propagate and catch errors due to physical defects and also to verify the effect on the connectivity measure. As expected, the connectivity of the enhanced RTOS (last line of Table 2.28) increases a little more than 1%, reaching a value of 76.91%. The absolute low value of connectivity is mainly motivated by the high number of branches in the RTOS code, due to the verification points concentrating on data and control flow. Nevertheless, the increase in the connectivity metric of SLT application may also lead to an increase in the fault coverage of the same test program.

2.4.12 Conclusions and Future Works

Fault coverage for functional programs has become more complex and time-consuming with the advent of complex SoCs. Nevertheless, the industry uses more and more functional tests. As a consequence, there is a rising necessity to abate the evaluation process cost. Based on the dynamic taint analysis paradigm, the proposed methodology recognizes critical edges in the control and data flow graph of functional code. Starting from the instruction sequence generated by the SoC, we first build a graph, and then we analyze it. The returned results could guide test engineers in

developing fault coverage effective programs. The proposed approach early and quickly highlights potential issues that can impact the fault coverage. The defined connectivity metric helps in both automated and manual generation flows to reduce fault simulation runs and guide random-based test program generation. Moreover, as the instruction sequence is obtained in seconds, we may be orders of magnitude faster than the standard fault simulation. Our results highlight an exciting correlation between the connectivity metric and the fault coverage of functional test programs.

2.5 Conclusions

In synthesis, in these years we worked for optimizing and speeding up the testing process. First, by creating a toolchain for analyzing VCD files. Then, providing a new metric for the analysis of test programs performed before the execution of the full fault simulation that can help catching obvious mistakes.

One of the lessons we can learn is that writing programs in machine-code compiled programming languages such as C, C++ and Rust does impact on the feasibility of analyses of large data, despite what most software engineers may say.

However, methodologies count too: while exact methods provide the best analysis types, significant results can be obtained by less accurate analyses, provided that they are correlated with the desired metric.

This chapter is at its end, and in the next one we will face my works on algorithms.

Chapter 3

Algorithms

In this chapter, I will talk about the works available in [65], in which we describe a parallel implementation of a graph coloring algorithm called JPL. This work has been developed and published together with the student Alessandro Borione and my colleague Lorenzo Cardone. Then, I will present the work available in [66], where we describe an improvement over the state-of-the-art algorithm McSplit and its variants for calculating the Maximum Common Subgraph. This work has been developed together with the students Salvatore Licata and Marco Porro, and my colleague Lorenzo Cardone.

3.1 Graphs

A graph is a pair of vertices (nodes) and edges (links). Links represent connections with nodes, making this structure well-suited for representing relationships between objects. In our notation, we use G and H to represent two graphs and $V(G)$ ($V(H)$) to represent the vertices belonging to G (H). Furthermore, we use $E(G)$ (and $E(H)$) to represent the set of all the pairs of vertices connected by an edge. We use $|G|$ or $|V(G)|$ to indicate the number of vertices belonging to G , referring to it as its *size*. In contrast, we refer to the number of edges of a graph as $|E(G)|$. Given $v_1, v_2 \in V(G)$, we denote $E(v_1, v_2)$ the edge that links v_1 to v_2 .

Graphs can come in various flavors: Labeled or unlabeled, weighted or unweighted, directed or undirected. In labeled graphs, vertices have additional informa-

tion described by the label; in many applications, the labels *classify* the vertices as sharing specific characteristics. In our notation, $L(v)$ is the label of the vertex v .

We say that the graph is *weighted* if edges present different weights associated with them. For example, a weight might represent the distance between two nodes. Unweighted graphs can be seen as weighted graphs with every weight equal to one.

We say that G is *undirected* if

$$\forall v_1, v_2 \in V(G) \in E(G) \iff \{v_2, v_1\} \in E(G) \ \& \ E(v_1, v_2) = E(v_2, v_1)$$

In other words, if a link exists between v_1 and v_2 , the opposite link must exist and have the same weight.

3.1.1 Notation

We refer to a graph G as $G = (V, E)$, where $V = \{v_1, v_2, \dots, v_n\}$ is the set of vertices, and $E \subseteq V \times V$ the set of edges. More specifically, we will manipulate undirected graphs, where an edge $e \in E$ is an unsorted pair of vertices

$$(v, u) \in E \iff (u, v) \in E \quad \forall v, u \in V$$

We also refer to the cardinality of V and E with n and m , respectively. Moreover, we use the notation

$$adj(v) = \{u \mid (u, v) \in E\}$$

to indicate the adjacency list of v , i.e., the set of nodes that share an edge with v .

Given a graph G , an independent set of nodes I is defined as:

$$I = \{v, u \mid (v, u) \notin E, \forall v, u \in V, v \neq u\}$$

that is, the independent set I includes vertices that are not adjacent. A maximal independent set is an independent set that is not a subset of a larger independent set.

In our implementations, we store a graph G adopting the so-called *Compressed Sparse Row* (CSR) representation. CSR is particularly efficient when large and sparse graphs must be represented since it is a matrix-based representation that stores

only non-zero elements of every row. Using this strategy, we can offer fast access to the information related to each row, avoiding useless overhead for very sparse matrices at the same time. Essentially, in the CSR format, edges are represented as a concatenation of all adjacency lists of every node. One additional array is used to index the adjacency list of each vertex in the main array. It is also possible to use a third array to store information about the weight of each edge. We will not use this additional array since we work with non-weighted graphs. Figure 3.1b reports the CSR representation for the graph of Figure 3.1a. As an example, in order to iterate through the neighbors of node 1, we would have to access the elements of the adjacency list array starting from the node at position 3 and ending at position 4, since the next node would start at position 5. In general, $adj(v)$ is the sub-array of the adjacency list array starting from $indexes[v]$ included, and ending at $indexes[v + 1]$ excluded.

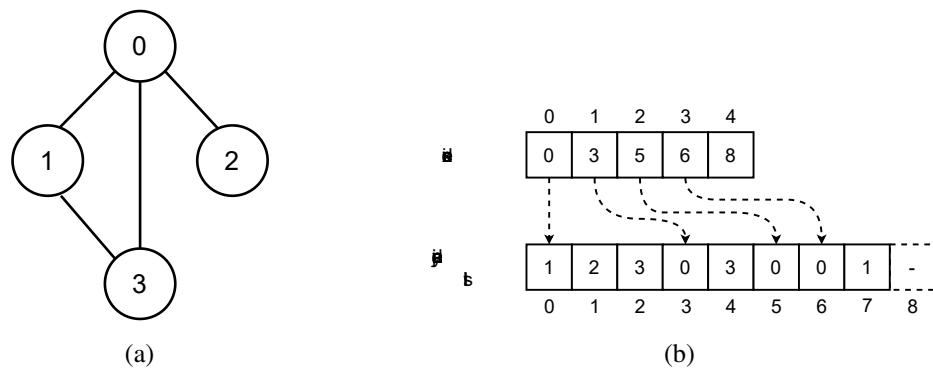


Fig. 3.1 An example of the Compressed Sparse Row (CSR) format: Graph (a) and corresponding CSR representation (b).

3.2 Graph Coloring on GPU

3.2.1 Introduction

The rapid accumulation of massive graphs from a diversity of disciplines, such as social and biological networks, geographical navigation, Internet routing, databases, and XML indexing, among others, requires fast and scalable graph algorithms.

Graph coloring is one of the many problems applied to graphs that benefit from algorithms that can produce reasonable solutions quickly. Graph coloring is useful

in many different fields, such as timetable scheduling [67, 68], register allocation in compiler optimization [69], Sudoku solving [70], parallelization of tasks [71], and many others. Graph coloring aims to assign a label (i.e., a color) to every vertex of the graph, such that adjacent vertices never have the same label. The problem of generating the best solution, i.e., the solution with the least number of labels, is known to be NP-hard [72]. Luckily, many applications that benefit from graph coloring do not strictly require an optimal solution, and a good approximation is often enough. As a consequence, many scalable heuristics [73–76] have been proposed over the years to approximate the perfect coloring in a reasonable time. However, heuristics typically present a tradeoff between the time to find a solution and the quality of the coloring process such that very often, the fastest heuristics produce the worse results and vice-versa. Moreover, data is being produced and collected faster, encompassing more information than ever, and graphs are also getting larger, causing the need for scalable algorithms to keep up with the times. Thus, researchers are focusing on solving graph coloring on huge graphs, derived from current data, in a faster and more scalable way.

Interestingly, general-purpose computing on GPUs (Graphical Processing Units) is increasingly used to deal with computationally intensive algorithms coming from several domains [77, 78]. The advent of languages such as OpenCL and CUDA has transformed GPUs into highly-parallel systems which scale gracefully, have a considerable bandwidth, and possess enormous computational power. As a consequence, there has been a continuous effort to redesign graph algorithms to exploit GPUs and CUDA, both from NVIDIA, with its NVIDIA Graph Analytic library, and from independent projects, such as the recent Gunrock library.

We compare our versions against two state-of-the-art GPU implementations, NVIDIA’s cuSparse library [76] and the Gunrock framework [79], and against a task-based approach for solving graph-related problems, i.e., Atos [80]. We present the number of colors used and the time required to perform the pre-processing, coloring, and post-processing phases on publicly available benchmarks. When we concentrate on the core (coloring) phase, we illustrate that our fastest implementation presents geomean (harmean) speedups of 3.16x (3.05x) against Gunrock, 4.09x (3.06x) against cuSparse, and 4.45x (2.21x) against Atos on graphs with low average degree. Nonetheless it is slower on scale-free graphs and ones with high average degree, presenting geomean (harmean) speedups of 2.76x (2.71x) against Gunrock, 0.13x (0.11x) against cuSparse, and 0.03x (0.01x) against Atos. When we concentrate on

the entire process (pre-processing, processing, and post-processing phases, including transfer times), we present geomean (peak) speedups of 7.43x (61.07x) against Gunrock. Moreover, our fastest technique produces 47% fewer colors than cuSparse and 7% fewer colors than Gunrock, based on the same JPL approach. Furthermore, it generates 63% more colors than Atos based on the GM approach, which is known to be slower but produces better coloring results.

3.2.2 Graph Coloring

Given a graph G , the target of graph coloring is to assign a color $color(v)$ to every vertex $v \in V$, such that if $u \in adj(v)$ then $color(v) \neq color(u)$. It is worth noting that the graph coloring problem is well-defined only for undirected graphs; given that no pair of adjacent vertices can have the same color, it is required that the property of being adjacent is symmetric. In other words, if vertex v is adjacent to vertex u , vertex u must also be adjacent to vertex v . For this reason, we run our experiments on graphs that are either undirected or are directed but have been pre-processed to double all their edges.

The classical approach to graph coloring sequentially visits all vertices $v \in V$, and assigns to each of them the color identified by the lowest number not yet assigned to its neighbors. Algorithm 8 reports the pseudo-code of this greedy approach. The quality of the solution depends on the order in which the nodes are considered. There exists a specific ordering that generates the optimal solution with the least number of colors possible, but finding this ordering is NP-hard [72]. Different heuristics have been proposed as approximate orderings. For example, the Largest-Degree First (LDF) heuristic [68], which colors vertices in order of decreasing degree, usually produces surprisingly good results. Unfortunately, albeit very simple, the greedy algorithm is inherently sequential and difficult to parallelize without major modifications.

We developed two versions of the Gebremedhin-Manne and one of Jones-Plassmann-Luby algorithm for multi-core CPUs. We also present two versions of the Jones-Plassmann-Luby approach for many-core GPU architectures.

We compare our versions, architecturally and experimentally, with the `csrColor` routine from the cuSparse library [81], the graph coloring program distributed with the Gunrock library [82], and the task-based approach implemented in Atos [80].

Algorithm 8 Greedy Graph Coloring example

GREEDY ($G = (V, E), colors$)
 1: $V' \leftarrow V$
 2: **for** $i = 1$ to n **do**
 3: Choose a vertex v_i from V'
 4: $color(v_i) \leftarrow \min c \in \mathbb{N} \setminus \{color(u) \mid u \in adj(v_i)\}$
 5: $V' \leftarrow V' \setminus \{v_i\}$
 6: **end for**

We analyze these algorithms in the following sections.

3.2.3 Jones-Plassmann-Luby

Luby [74] suggests that an independent set of nodes can be colored in parallel with the same color without conflicts, and develops an algorithm to find maximal independent sets in a graph.

Jones and Plassmann [75] develop Luby's approach using independent sets. Their strategy finds non-maximal independent sets by assigning a random number to each vertex, and selecting all nodes whose random numbers are local maxima. Each node in the independent set is then colored separately in parallel with the lowest color not assigned to one of their neighbors.

We define the Jones-Plassmann-Luby procedure (JPL) as a middle ground between Jones-Plassmann and Luby's algorithm. First, we find non-maximal independent sets using the random values approach from the Jones-Plassmann algorithm. Then, we color each independent set with a single color like Luby's algorithm. The process is repeated until all nodes are colored. The JPL procedure is reported in Algorithm 9. First, every node v is assigned a random number $\rho(v)$. Then, the algorithm iterates until all nodes are colored. At each step, i of the iteration, an independent set I_i is computed using random numbers. More specifically, a node v is part of I_i if and only if it is yet to be colored, and every one of its neighbors $u \in adj(v)$ is assigned a random value $\rho(u)$ so that $\rho(v) > \rho(u)$. We can say that, if $v \in I_i$, v is a local maximum because its assigned random value is a maximum in the locality of its (non-colored) neighbors. The iteration ends when all members of I_i are assigned with the color i .

Algorithm 9 Jones-Plassmann-Luby coloring heuristic

```

JPL-COLOR ( $G = (V, E)$ )
1:  $N \leftarrow V$ 
2:  $i \leftarrow 1$ 
3: for  $v \in N$  do
4:    $\rho(v) \leftarrow$  random number
5: end for
6: while  $N \neq \emptyset$  do
7:    $I \leftarrow \emptyset$ 
8:   for  $v \in N$  in parallel on GPU do
9:      $I \leftarrow I \cup \{v\}$ 
10:    for  $u \in (adj(v) \cap N)$  do
11:      if  $\rho(v) \leq \rho(u)$  then
12:         $I \leftarrow I \setminus \{v\}$ 
13:      end if
14:    end for
15:  end for
16:  for  $v \in I$  in parallel do
17:     $color(v) \leftarrow i$ 
18:  end for
19:   $N \leftarrow N \setminus I$ 
20:   $i \leftarrow i + 1$ 
21: end while

```

Following this logic, Algorithm 9 is divided into two main loops. The first iteration (starting at line 3) contains the initialization of the random values associated with each node in the graph. The second cycle colors the graph, and it is divided into two more sections, i.e., the computation of an independent set (lines 8–15), and the actual coloring (line 17). The introduction of independent sets allows coloring nodes in parallel without risking inconsistencies in coloring adjacent nodes. However, this procedure quickly computes each independent set, but then colors each one of them independently; thus, it may require a large number of iterations. This situation presents itself when sets of nodes with increasing random identifiers are adjacent and form a linear sequence. For example, let us suppose that a node with label 1 is adjacent to a node with label 2, which, in turn, is adjacent to a node with label 3. As a consequence, only one node per iteration will be colored, as node 3 will be colored first, followed by node 2 during the second iteration, and node 1 during the last loop.

Figure 3.2 shows the JPL algorithm applied to an example graph. Each random number $\rho(v)$ is displayed inside the corresponding node v that is not colored. For the sake of simplicity, we use random integer numbers between 0 and 99. Figures 3.2b,

3.2c, 3.2d, 3.2e, and 3.2f show the state of the coloring after each iteration. In Figure 3.2a, when all nodes are yet to be colored, we can notice the linear sequence of nodes with random numbers $78 \rightarrow 62 \rightarrow 57 \rightarrow 40 \rightarrow 13$, and how the length of the chain (5) is equal to the number of iterations required to complete the coloring.

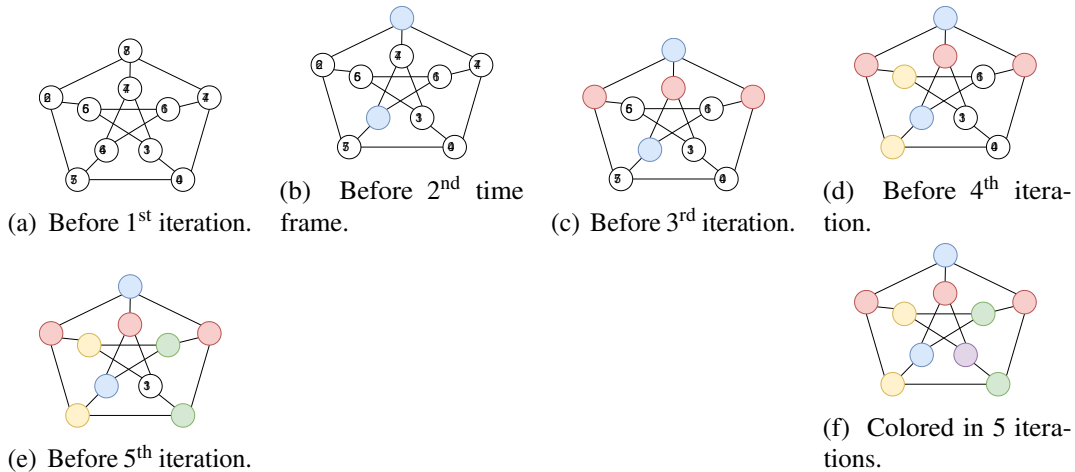


Fig. 3.2 Application of the Jones-Plassmann-Luby algorithm on a small graph of 10 nodes.

3.2.4 Gebremedhin-Manne

Gebremedhin and Manne [73] propose a parallel graph coloring algorithm whose core idea is to allow inconsistencies in the coloring process. Specifically, they first divide the vertices into p blocks. Then, they mock-color the vertices of all blocks in parallel. We use the term “mock-color” because the resulting coloring may present a conflict every time two (or more) adjacent vertices are colored by two (or more) different threads simultaneously. Thus, the mock-coloring phase is followed by a parallel phase to discover all conflicts and a final sequential phase where the conflicts are rectified. They also present an improved version of the same algorithm to reduce the number of colors generated during the mock-coloring step.

Algorithm 10 and Algorithm 11 show the standard and an improved algorithms, respectively.

Algorithm 10 is formed by three sections, each corresponding to one for-loop. In the first part (line 1), every graph node is colored in parallel, allowing for coloring errors, i.e., two adjacent nodes can be assigned the same color. In the second part (line 6), the errors generated in before are found. Each pair of neighboring nodes

is checked in parallel so that conflicting pairs are saved to be managed later. The number of pairs to check can be reduced by only considering pairs of nodes that were colored at the same time frame during the first part of the process. Time frames, or steps, are a consequence of using a barrier in the first part. As only the nodes that are colored at the same time can present a conflict in the colors assigned, and race conditions between the working threads cause these conflicts, the authors suggest to reduce the pairs of neighbors to avoid coloring inconsistencies. In Algorithm 10, this is shown in the set of nodes S colored in the same step as the current node on line 7, and on the intersection $adj(v) \cap S$ on line 8. Thus, K represents the nodes that need to be recolored. In the third part (line 14), the nodes that were in conflict are recolored, this time sequentially, to avoid adding more coloring errors that would need to be discovered and corrected in the same way. The standard algorithm is relatively slow, as the slowest thread, i.e., the one that colors the node with the most neighbors each time frame, blocks the other threads on the barrier synchronization on line 3. Experimentally, this is shown to take up between 80% and 90% of the execution time.

Algorithm 10 Gebremedhin-Manne Standard Algorithm.

```

GEBREMEDHIN-MANNE-STANDARD ( $G = (V, E), colors$ )
1: for  $v \in V$  in parallel do
2:    $colors(v) \leftarrow \min c \in \mathbb{N} \setminus \{colors(u) \mid u \in adj(v)\}$ 
3:   Barrier wait
4: end for
5:  $K \leftarrow \emptyset$ 
6: for  $v \in V$  in parallel do
7:    $S \leftarrow$  nodes colored in the same step as  $v$  in line 2
8:   for  $u \in (adj(v) \cap S)$  do
9:     if  $colors(v) = colors(u)$  then
10:       $K \leftarrow K \cup \min\{v, u\}$ 
11:     end if
12:   end for
13: end for
14: for  $v \in K$  do
15:    $colors(v) \leftarrow \min c \in \mathbb{N} \setminus \{colors(u) \mid u \in adj(v)\}$ 
16: end for

```

Figure 3.3 shows a simple graph being colored following Gebremedhin-Manne standard formulation using two blocks. Nodes with the same border color (red or green) belong to the same block. We also assume that processors color the nodes based on the status of the previous time frame, without entering race conditions

within the same frame. This assumption is a simplification, as the actual behavior depends on how the processors are scheduled at runtime. Nodes within a block are colored in lexicographical order, which we assume is clockwise, outer to inner, starting from the top-most node. We report the state of the coloring after each time frame in Figures 3.3b, 3.3c, 3.3d, 3.3e, and 3.3f. Figure 3.3g shows the state after the conflict search step, which only detects a single conflict. Lastly, Figure 3.3h reports the final coloring after the conflict correction step is completed. The solution produced uses three colors, which we know is the lower bound to the number of colors for this graph.

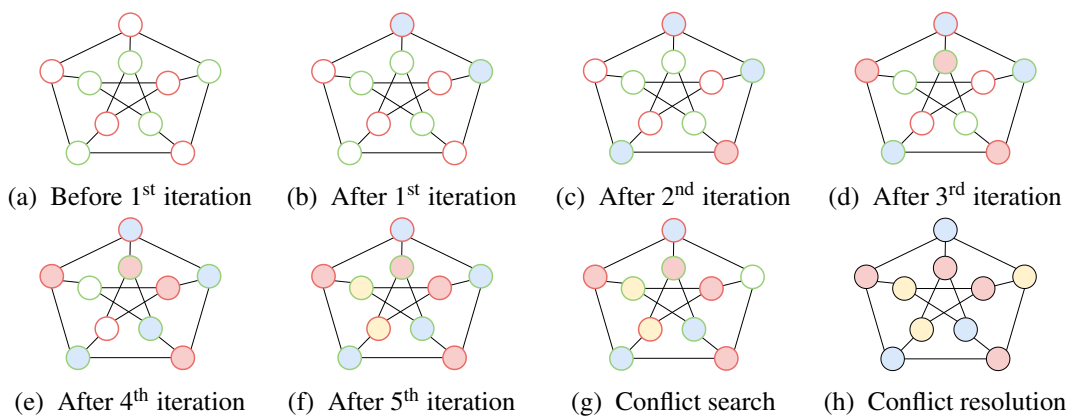


Fig. 3.3 Application of the Standard Gebremedhin-Manne algorithm on a 10 nodes graph

Algorithm 11 tries to reduce the number of colors produced by Algorithm 10. It can be logically divided into four parts. Parts one, three, and four are equivalent to the ones appearing in the standard algorithm, in parts one, two, and three, respectively. Part two performs a second coloring, still allowing for mistakes. The coloring is performed by first dividing the nodes into color classes. A color class is the set of nodes assigned the same color in part one of the algorithm. Then the coloring from part one is deleted, and all nodes are recolored in parallel, starting from the ones belonging to the color class that was assigned the largest color in part one, and continuing with the color classes assigned with lower colors, until all nodes are colored a second time. At the end of part four, the improved algorithm finds a solution using a number of colors lower or equal to the number of colors used during the first coloring at the end of part one [73]. Despite producing better coloring, the improved algorithm is slower than the standard algorithm, as the coloring is performed twice. The conflicts after part two are fewer than with the standard algorithm, as shown empirically in the original research.

Algorithm 11 Gebremedhin-Manne Improved Algorithm

GEBREMEDHIN-MANNE-IMPROVED ($G = (V, E), colors$)

- 1: $colors' \leftarrow colors$
- 2: **for** $v \in V$ in parallel **do**
- 3: $colors'(v) \leftarrow \min c \in \mathbb{N} \setminus \{colors'(u) \mid u \in adj(v)\}$
- 4: Barrier wait
- 5: **end for**
- 6: **for** k from $\max_{v \in V} colors'(v)$ down to $\min_{v \in V} colors'(v)$ **do**
- 7: $ColorClass \leftarrow \{v \in V \mid colors'(v) = k\}$
- 8: **for** $v \in ColorClass$ in parallel **do**
- 9: $colors(v) \leftarrow \min c \in \mathbb{N} \setminus \{colors(u) \mid u \in adj(v)\}$
- 10: **end for**
- 11: Barrier wait
- 12: **end for**
- 13: $K \leftarrow \emptyset$
- 14: **for** $v \in V$ in parallel **do**
- 15: **for** $u \in adj(v)$ **do**
- 16: **if** $colors(v) = colors(u)$ **then**
- 17: $K \leftarrow K \cup \min\{v, u\}$
- 18: **end if**
- 19: **end for**
- 20: **end for**
- 21: **for** $v \in K$ **do**
- 22: $colors(v) \leftarrow \min c \in \mathbb{N} \setminus \{colors(u) \mid u \in adj(v)\}$
- 23: **end for**

The two previous procedures can be considered as “synchronous” as all threads wait on a barrier (Algorithm 10 on line 3, and Algorithm 11 on lines 4 and 11) before proceeding to the next iteration. Unfortunately, in these algorithms, 90% of the time is spent waiting on the barriers. Thus, Algorithm 10 can be converted to an asynchronous version by removing the barrier wait synchronization on line 3; consequently, line 7 is substituted with $S \leftarrow V$ as, without the barrier, there are no more time frames. Similarly, Algorithm 11 can be made asynchronous by removing the barrier wait call on lines 4 and 11. The asynchronous formulations run faster than their synchronous counterpart as the bottleneck on the barrier synchronization is removed. However, the coloring produced by the asynchronous algorithm is more affected by the execution schedule and generally uses more colors than the synchronous algorithm.

In their paper, Gebremedhin and Manne propose to divide the n nodes to be colored in p blocks of n/p nodes. Each block V_i is then assigned to a processor p_i , with $(1 \leq i \leq p)$, that works on that block alone. The paper fails to address how the nodes are distributed across the blocks. The chosen distribution rule determines how, in cases where (n/p) is not an integer, the remaining $(n \bmod p)$ nodes are distributed into the partitions. In our implementation of the algorithms, we decided to assign the nodes to the blocks based on their lexicographical ordering. The first p nodes are assigned one to each block, then the second p nodes, and so on. Each block V_i is composed of the nodes:

$$\{v_{k*p+i} \mid 1 \leq k \leq \lceil n/p \rceil\}$$

like in the example of Figure 3.3a. With our distribution policy, if (n/p) is not an integer, we consider $(n \bmod p)$ more nodes that are not part of the original set V . We call these “ghost” nodes, and they are distributed so that all blocks are of the same size $\lceil n/p \rceil$. Ghost nodes are dummy nodes and do not belong to the original graph, so they do not need to be colored. Instead, in the synchronous versions of the algorithm, they serve the purpose of keeping all processors inside the first for-loop (Algorithm 10 line 1 and Algorithm 11 line 2). In this way, all processors leave the for-loop after $\lceil n/p \rceil$ iterations, and there is no need to resize the barrier to accommodate processors that would exit earlier. For the sake of brevity and simplicity, we did not add the block-separator steps in Algorithm 11, since the algorithm itself still implies that the concurrency is limited by the number of processors.

3.2.5 Atos

Chen et al. [80] propose Atos, i.e., a parallel task-based methodology applicable to GPUs that derives from the GM approach. Atos proposes to run a pair of GPU kernels split over two different algorithms. The first kernel works on a frontier consisting of the non-colored nodes and assigns a color to it in a Gebremedhin-Manne-like manner. In practice, the kernel selects a color based on the node's neighborhood, and then adds the node to the frontier of the second kernel. The second kernel checks the correctness of the assignment, making sure that two adjacent nodes never have the same color. It then consumes the nodes newly colored by the first kernel and treats them differently based on the success or failure of the check. In the case of no conflict, the node is removed from the frontier and is permanently colored; on the contrary, in the case of a conflict the node is reassigned to the frontier of the first kernel, forcing it to a new coloring phase. The process continues until all nodes have been permanently colored and no conflict is present anymore. The approach has been designed to prevent the two main problems of Bulk Synchronous Parallel algorithms, i.e., programs that fully utilize the GPU launching a single GPU-wide kernel. The two problems are Load Imbalance and Small Frontier. The former of the two occurs when some threads are faced with significantly more computations than others. The latter happens when there are fewer processes available than the number of threads available.

3.2.6 Cohen-Castonguay

Cohen and Castonguay [76] present a GPU-based algorithm for graph coloring derived from the JPL algorithm. They suggest three critical modifications to the original algorithm.

The first suggestion consists in improving the number of nodes that can be concurrently colored at each iteration. To maintain the efficiency of the original algorithm, they propose a way to select two independent sets that are disjointed with little or no extra complexity. In each iteration, the authors search the set of local maxima I_i^M , as shown in Section 3.2.3, and the set of local minima I_i^m . These two sets are independent, which means that there are not adjacent node pairs belonging to the same set. This property holds because it is not possible for two adjacent nodes to simultaneously share the property of having the highest (for the set I_i^M) and the

lowest weight (for the set I_i^m) weight among all their neighbors. Moreover, except for nodes with no neighbors, the two sets are also disjoint since it is not possible for a node with neighbors to possess a weight simultaneously larger and smaller than the weights of every other adjacent node. These two properties allow us to color the two sets in parallel with two distinct colors in a single pass [83]. Furthermore, they prove that a parallel algorithm cannot select more than two disjoint independent sets per iteration in the JPL function [83].

The second modification is based on the observation that the vector of random values resides in memory, and each access to one of its element is inherently slow. They suggest that it is possible to disregard the necessity to have the vector of random values by using the hash function $H : V \rightarrow K$, where K is a general set where its members can be ordered. A hash function can compute seemingly random values if given the node identifiers as input. The results can be maintained in registers for fast access, then discarded, and recomputed on the fly when needed again. Even if each number must be recomputed several times, the strategy is faster than accessing the value in the main memory.

The third modification is based on the idea of using k hash functions H_1, \dots, H_k , thus finding more than two sets per iteration. It must be noted that a function H_i generates two disjoint independent sets, but the two sets, generally, are not disjoint from the two sets generated by another function H_j . Thus the hash functions must be sorted, and a lower-ranking function can consider only nodes not colored by the higher-ranking functions. By using k hash functions, it is possible to color $2k$ sets at a time, significantly reducing the number of iterations needed to color the whole graph. However, the number of hash functions k must be carefully chosen. Having too many hash functions may reduce the speed of the algorithm and hinder the quality of the solution as overlapping independent sets need to be made disjoint before coloring, adding overhead to the algorithm.

These observations make the algorithm perfect for SIMT architecture, where the main bottlenecks are often the memory bandwidth and global synchronization needed to run an algorithm like JPL. Since the hash functions implemented in the algorithm do not change, Cohen-Castonguay is a deterministic algorithm, meaning that given the same input, the output will always be the same as well. Among the algorithms that we considered, this is the only one to have this property intrinsically implemented. The Jones-Plassmann-Luby algorithm can also be adapted to ensure a

deterministic result by using a seed for the random number generation. Similarly, Gunrock and our implementation, based on JPL, can be modified accordingly. The Cohen-Castonguay algorithm is made available through the `csrColor` routine of the `cuSparse` library [81].

3.2.7 Gunrock

Gunrock [82] is an open-source library designed to solve graph processing problems on CUDA-enabled GPUs. Gunrock is distributed with a wide variety of graph primitives, among which there is a graph coloring primitive. Gunrock’s graph coloring algorithm follows the research by Osama et al. [79]; the implementation, described in Algorithm 12, follows a variation of the JPL algorithm. Similar to the Cohen-Castonguay algorithm presented in Section 3.2.6, Gunrock searches for two independent sets per iteration. However, it retains the vector of random values, called *rand* in Algorithm 12. Moreover, their algorithm does not need any form of load balancing that would imply conspicuous time overheads. Their tests show that their implementation is the fastest one on GPUs.

In our experiments, we adopted the Gunrock library within the development branch dated 15 November 2021. This version of the coloring algorithm has a flaw [84], causing infinite loops and preventing the program from finishing with a correct solution. The problem is caused by the management of the *rand* vector. Initially, the vector is populated with random, single-precision floating point values. In lines 15 and 18 of Algorithm 12, the process compares two values of the vector to choose the node v to remove from the corresponding independent set. However, if $rand(v) = rand(u)$, nodes v and u are both removed from both sets by their respective threads. As a consequence, both v and u are never part of an independent set, meaning that they will never be colored, thus leading to an infinite loop as the algorithm terminates when all nodes are colored.

We propose two different approaches to fix this problem.

Our first approach follows the observation that a simple tie-breaking condition would solve the issue when comparing the two values. Thus, we substitute the conditional on line 15 with $rand(v) < rand(u)$ **or** $(rand(v) = rand(u) \ \& \ v < u)$ and the conditional on line 18 with $rand(v) > rand(u)$ **or** $(rand(v) = rand(u) \ \& \ v > u)$. In this new version, the tie is broken with a comparison of the two node indexes,

Algorithm 12 Gunrock coloring procedure.

```

GUNROCK-COLOR ( $G = (V, E), rand, colors$ )
1:  $i \leftarrow 0$ 
2:  $N \leftarrow V$ 
3: while  $N \neq \emptyset$  do
4:   if  $i \bmod 2 = 0$  then
5:     for  $v \in N$  in parallel on GPU do
6:        $rand(v) \leftarrow$  random number
7:     end for
8:   end if
9:    $c \leftarrow 2 * i + 1$ 
10:   $k \leftarrow 2 * i + 2$ 
11:  for  $v \in N$  in parallel on GPU do
12:     $I^M \leftarrow I^M \cup \{v\}$ 
13:     $I^m \leftarrow I^m \cup \{v\}$ 
14:    for  $u \in (adj(v) \cap N)$  do
15:      if  $rand(v) \leq rand(u)$  then
16:         $I^M \leftarrow I^M \setminus \{v\}$ 
17:      end if
18:      if  $rand(v) \geq rand(u)$  then
19:         $I^m \leftarrow I^m \setminus \{v\}$ 
20:      end if
21:    end for
22:  end for
23:  for  $v \in I^M$  in parallel on GPU do
24:     $colors(v) \leftarrow c$ 
25:  end for
26:  for  $v \in (I^m \setminus I^M)$  in parallel on GPU do
27:     $colors(v) \leftarrow k$ 
28:  end for
29:   $N \leftarrow N \setminus I^M$ 
30:   $N \leftarrow N \setminus I^m$ 
31:   $i \leftarrow i + 1$ 
32: end while

```

which are unique by definition. In other words, a node v is considered a local maximum if $rand(v) \geq rand(w) \ \& \ v > u, \forall w \in N, \forall u \in M$, where $N = \{w_1, \dots, w_k\}$ is the set of the non-colored nodes adjacent to v , and $M = \{u_1 \dots u_b\} \subseteq N$ is the subset of N where $rand(v) = rand(u_i)$. Similarly, a node v is considered a local minimum if $rand(v) \leq rand(w) \ \& \ v < u, \forall w \in N, \forall u \in M$.

Our other approach takes inspiration from previous versions of the coloring primitives where the *rand* vector had its values regenerated every other iteration. This approach fixes the issue because if v and u are assigned the random values $rand_i(v) = rand_i(u)$ at iteration i , it is expected that at iteration $j \geq i + 2$, $rand_j(v) \neq rand_j(u)$. In this way, two adjacent nodes can share the random maximum or minimum at some point, but will be colored at a later iteration, when the random values are eventually different. We get confirmation from the Gunrock developers that this second approach enables the intended behavior of their tool [84]. In Section 3.2.9, we refer to the Gunrock implementation complete with this approved fix. The regeneration happens in the block on line 4 of Algorithm 12.

In Figure 3.4, we report an example of a graph colored with the JPL implementation from Gunrock. The graph used and the associated random numbers are the same as adopted in Figure 3.2, to simplify the comparison of the two algorithms. Figures 3.4b, 3.4c, and 3.4d show the state of the graph after each iteration of the algorithm. The chain of nodes with random numbers $78 \rightarrow 62 \rightarrow 57 \rightarrow 40 \rightarrow 13$ is colored from both ends simultaneously, thus reducing the number of iterations needed to complete the coloring. Moreover, after 2 iterations, the random numbers of the nodes are recomputed. Figure 3.4c shows the state of the graph before starting the 3rd iteration, after the recomputation is finished; however, only one node remains to be colored. The recomputation is inconsequential in this small example, but it helps in reducing the number of colors and iterations on larger graphs. The graph is colored using the same number of colors as in Figure 3.2, but using half the number of iterations rounded up.

3.2.8 Our Coloring Procedure

Our implementation follows the Jones-Plassmann-Luby algorithm described in Section 3.2.3. We designed our code to be compiled in two versions: One finding a single independent set for each iteration, and the other finding two independent

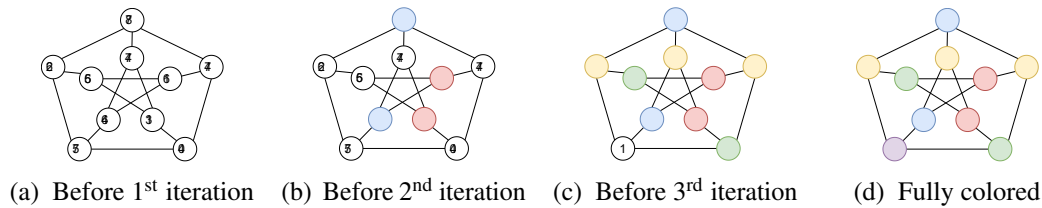


Fig. 3.4 Application of the Gunrock implementation of the JPL algorithm on a 10 nodes graph

sets. The two versions enable us to verify the speedup reported by Osama et al. [79], that can be achieved finding two independent sets per iteration. We will show in Section 3.2.9 that our speedup of the Min Max approach against the Max approach ranges between 1.5X and 2X on average. We declare the kernel function `COLOR_JPL`, which takes as input the entire graph in CSR format, the pre-initialized array of random values *rand*, and the array where to store the colors. The total memory occupancy on the GPU is of $(4 \times (3n + m + 1))$ bytes, assuming the architecture uses 32 bit integers. When working with graph instances that do not fit in the limited memory of the GPU, we partition the nodes of the graph in lexicographical order so that they fit in memory; thus, we perform multiple colorings to color each partition separately. The pseudo-code for the kernel function is shown in Algorithm 13.

To solve the problem of two neighboring nodes being assigned the same random value, as described in Section 3.2.7, we proceed as follows. In the first phase, to obtain the array *rand* received as a parameter by the function `COLOR_JPL`, we pre-generate a random permutation of a set of unique items such as V . We call this technique **value permutation**. Generating permutations is a simple yet powerful way to solve the problem of two neighboring nodes being assigned the same random value. Thus, our *rand* array contains a permutation of V , unlike Gunrock's *rand* array, which includes random floating point numbers. To perform this task, we initially adopted the randomization function `std::shuffle`¹ available in the C++ standard library. Unfortunately, even though this function has linear complexity in the size of the array (i.e., the number of nodes in the graph), a close investigation of the entire execution time (not just the coloring phase), showed us that the computational overhead on large graphs could obfuscate the advantage of our approach. Although the array generation requires only the number of nodes of the graph to be performed, and its time could be entirely masked by other algorithmic phases performed in

¹https://en.cppreference.com/w/cpp/algorithm/random_shuffle

Algorithm 13 Our implementation of the Jones-Plassmann-Luby algorithm

```

COLOR_JPL ( $G = (V, E)$ ,  $rand$ ,  $colors$ )
1:  $i \leftarrow 0$ 
2:  $N \leftarrow V$ 
3: while  $N \neq \emptyset$  do
4:    $c \leftarrow 2 * i + 1$ 
5:    $k \leftarrow 2 * i + 2$ 
6:   for  $v \in N$  in parallel on GPU do
7:      $I^M \leftarrow I^M \cup \{v\}$ 
8:      $I^m \leftarrow I^m \cup \{v\}$ 
9:      $r_v \leftarrow rand(v + i \pmod n)$ 
10:    for  $u \in (adj(v) \cap N)$  do
11:       $r_u \leftarrow rand(u + i \pmod n)$ 
12:      if  $r_v \leq r_u$  then
13:         $I^M \leftarrow I^M \setminus \{v\}$ 
14:      end if
15:      if  $r_v \geq r_u$  then
16:         $I^m \leftarrow I^m \setminus \{v\}$ 
17:      end if
18:    end for
19:  end for
20:  for  $v \in I^M$  in parallel on GPU do
21:     $colors(v) \leftarrow c$ 
22:  end for
23:  for  $v \in (I^m \setminus I^M)$  in parallel on GPU do
24:     $colors(v) \leftarrow k$ 
25:  end for
26:   $N \leftarrow N \setminus I^M$ 
27:   $N \leftarrow N \setminus I^m$ 
28:   $i \leftarrow i + 1$ 
29: end while

```

parallel (such as allocating, building, or loading the graph itself), we decided to investigate faster solutions. We reduced the generation time of the *rand* vector to a fraction of the original time by randomly generating the array using a custom *fast_rand* function. The *fast_rand* function is a multiply-with-carry pseudo-random number generator, that allows to produce sequences of pseudo-random values with very long period, by using simple integer arithmetic logic. The function generates a weaker scattering of the generated values but this feature does not decrease the quality of the solution. This consideration may raise the question of how much quality of the randomness we can give away to speed up the vector generation process while not losing the quality and speed of the color computation. This area may be interesting for a future study on the subject. Moreover, inspired by our fix of the Gunrock implementation, we also change the random values assigned to each node every iteration. As in the Gunrock approach, regenerating the random values has a positive impact on the number of colors, since it helps split long chains of monotonic random values that would need many iterations to be colored. To avoid the overhead required by the generation of new arrays, we simulate a new permutation by accessing the same array circularly. In this way, the array still contains the same values, but each vertex v is assigned a new “random” value $rand_k(v)$ after k shifts, and the regeneration cost is meager. Shifting a n -element array in the device’s global memory is an operation that requires synchronization between the threads of the grid, and programming a GPU to perform it requires some care. However, the random value of vertex v after k iterations, i.e., $rand_k(v)$, is the random value of vertex $(v + k \bmod n)$ after 0 iterations, i.e., $rand_0(v + k \bmod n)$. In other words, shifting an array by k positions to the left is equivalent to increment the index of the same amount k and performing the proper modulo operation to remain within the array’s bound. We call this technique **index shift**, as it simulates an array circular shift by manipulating the index used to access the array. To simulate subsequent circular left shifts, one after each iteration, we decide to increase by one the indexes used by each thread cumulatively to access the *rand* array. Shifting the index does not add any significant overhead to the computation, as we do not write nor move data in memory. This technique is reported in Algorithm 13 on lines 9 and 11. Further experiments show that while the shift itself is always effective in reducing the number of colors; in some situations, this reduction can be further optimized by a full regeneration of the vector of random weights. By investigating the nature of these cases, we discovered that the probability that two adjacent nodes share an arc is higher than the probability

that any pair of nodes share an arc. Although this proved to be valid only on some graphs, the non-correlation between nodes with neighboring indices could not be taken for granted. Given the nature of the problem, we increase the shift in the vector, such that the probability that the two nodes separated by a number of elements equal to the shift have a relationship is reduced considerably. Precisely as in the case of shifting a single element, this operation adds almost no computational cost, since it only changes the memory access address.

Section 3.2.9 shows that our approach reduces the number of iterations needed to compute the final coloring. Consequently, our code runs faster and solves all our benchmarks with fewer colors than all previous implementations. Moreover, our experiments unveil that shifts greater than four rarely give any benefit. Although this value has been evaluated experimentally, its meaning is the following. Shifting the vector of random numbers changes the relationships between the nodes possessing the various weights. An extremely simple analysis, which can be performed at almost no additional cost when reading the file, is calculating the highest number of consecutive nodes that form adjacency chains. A shift of a length greater than this chain would allow us to avoid having the same random number assigned to another element of the same chain. However, this would still be a superficial analysis, since the values don't need to leave the chain in which they are located, as it is sufficient to vary the configuration of the adjacent nodes. Consequently, the size of the shift can be expressed as a function of the size of the graph and how dense or sparse it is.

Figure 3.5 shows how our implementation of the JPL algorithm colors the same small graph used in all other examples. The numbers we permute after each iteration, the ones stored in the array *rand*, are displayed inside their corresponding node v . The numbers are unique and included in the range of integers between 0 and 9. Figures 3.5b, 3.5c, and 3.5d show the state of the graph after each iteration of the algorithm. The unique numbers move according to the technique of index shifting. In the picture, we assume the nodes are ordered clockwise, outer to inner, starting from the top-most node. Each permutation moves those numbers corresponding to the circular left shift of the array. The final solution of Figure 3.5d is congruent with the one obtained by the original Gunrock implementation (and represented in Figure 3.4), but the algorithm runs faster and uses fewer colors, as we illustrate in the next section.

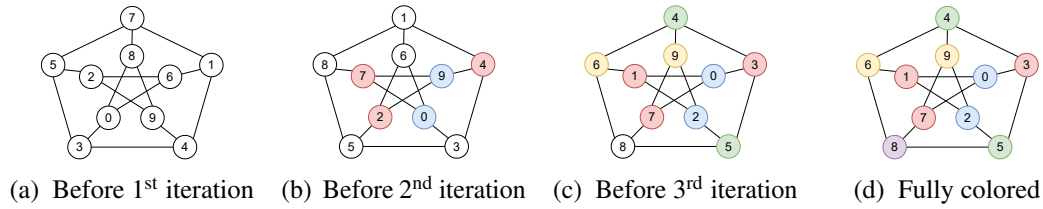


Fig. 3.5 Application of our implementation of the JPL algorithm on a 10 nodes graph

3.2.9 Experimental Results

We perform our experiments on an i9 10900KF CPU running at 3.7 GHz, with 10 cores, 20 threads, and 64 GB of RAM, coupled with a GPU NVIDIA RTX 3070 with 5888 CUDA cores and 8 GB of dedicated memory. The operating system is Linux Ubuntu 22.04.1 LTS. The code was compiled using Clang version 16.0.4 for the CPU implementations, and NVIDIA’s CUDA Compiler (NVCC) version 12.0 for the GPU programs.

We run the coloring implementations described in Sections 2.4.1 and 3.2.8 on the set of graphs reported in Table 3.1. The set contains the same graphs used by Osama et al. [79], plus some extra graphs, namely, email_Enron, twitch_gamers, qq.order100, hollywood-2009, indochina-2004, and soc-LiveJournal1. For each graph, the table reports the number of vertices and edges. Column *Type* indicates whether the graphs are real (**r**) or generated (**g**), and undirected (**u**) or directed (**d**). Moreover, the last part of the table includes graphs showing a power-law degree distribution (**p**). These five graphs marked with letter (**p**) are kept separated because the coloring algorithms have different behavior (and performances) on these instances. We gathered the graphs through the Sparse Matrix Collection website [85], if not otherwise specified.

If we consider the larger test graph, i.e., rgg_n_2_24_s0, with more than 16 million nodes, our implementation requires a GPU memory of

$$\begin{aligned}
 &4 \times (3n + m + 1) \\
 &= 4 \times (3 \times 16,777,216 + 265,114,400 + 1) \\
 &\approx 1.18 \text{ GB}
 \end{aligned}$$

As a consequence, memory is not an issue, as the most extensive graph occupies less than 15% of the total memory available in our GPU. To take into consideration runtime fluctuations and provide a better estimate of the number of colors used by all

#	Graph	Nodes	Edges	Avg degree	Type
1	af_shell3	504,855	17,588,875	34.8	ru-
2	apache2	715,176	4,817,870	6.7	ru-
3	ecology2	999,999	4,995,991	5.0	ru-
4	G3_circuit	1,585,478	7,660,826	4.8	ru-
5	offshore	259,789	4,242,673	16.3	ru-
6	parabolic_fem	525,825	3,148,801	6.0	ru-
7	thermal2	1,228,045	8,580,313	7.0	ru-
8	ASIC_320ks	321,671	1,827,807	5.7	rd-
9	atmosmodd	1,270,432	8,814,880	6.9	rd-
10	cage13	445,315	7,479,343	16.8	rd-
11	FEM_3D_thermal2	147,900	3,489,300	23.6	rd-
12	thermomech_dK	204,316	2,846,228	13.9	rd-
13	rgg_n_2_15_s0	32,768	320,480	9.8	gu-
14	rgg_n_2_16_s0	65,536	684,254	10.4	gu-
15	rgg_n_2_17_s0	131,072	1,457,506	11.1	gu-
16	rgg_n_2_18_s0	262,144	3,094,566	11.8	gu-
17	rgg_n_2_19_s0	524,288	6,539,532	12.5	gu-
18	rgg_n_2_20_s0	1,048,576	13,783,240	13.1	gu-
19	rgg_n_2_21_s0	2,097,152	28,975,990	13.8	gu-
20	rgg_n_2_22_s0	4,194,301	60,718,396	14.5	gu-
21	rgg_n_2_23_s0	8,388,608	127,002,786	15.1	gu-
22	rgg_n_2_24_s0	16,777,216	265,114,400	15.8	gu-
23	qg.order100[86]	10,000	1,980,000	198.0	gd-
24	twitch_gamers [87]	168,114	13,595,114	80.9	rup
25	email_Enron	36,692	367,662	10.0	rdp
26	hollywood-2009	1,139,905	112,751,422	98.9	rup
27	indichina-2004	7,414,866	301,969,638	40.7	rdp
28	soc-LiveJournal1	4,847,571	85,702,474	17.7	rdp

Table 3.1 The main characteristics of the benchmark graphs used during our experimental analysis. The graphs are numbered from 1 to 28 to find an easy correspondence in the following plots. Column *Type* indicates the main characteristics of each graph: Real (**r**) or generated (**g**), undirected (**u**) or directed (**d**), following a power-law degree distribution (**p**) or not (-). Notice that power-law graphs, which have distinct characteristics and on which the different approaches behave differently, have been inserted in the second part of the table.

non-deterministic algorithms, we run each implementation 20 times on each graph. Thus, our tables report the average time spent coloring the graph and the average number of colors used, rounded to the nearest integer.

#	Graph	CPU-based			GPU-based				
		GM_{s-imp}	GM_{a-std}	$JPL_{min-max}$	cuSparse	Gunrock	Atos	Our Methods	
								JPL_{max}	$JPL_{min-max}$
1	af_shell3	1387.70	63.39	29.31	5.97	28.09	12.15	10.14	5.63
2	apache2	1888.60	30.10	7.67	3.89	1.19	12.30	1.67	0.32
3	ecology2	2679.50	36.73	6.99	3.22	0.87	16.56	1.54	0.32
4	G3_circuit	4267.00	58.88	9.06	4.16	1.18	52.09	1.97	0.49
5	offshore	695.62	18.70	9.36	3.48	3.29	3.93	1.68	0.98
6	parabolic_fem	1371.00	22.76	5.94	2.79	1.06	0.84	1.22	0.43
7	thermal2	3285.30	59.97	9.37	4.52	2.03	3.83	2.00	0.88
8	ASIC_320ks	833.46	13.29	5.17	3.75	3.42	0.93	3.23	1.84
9	atmosmodd	3434.60	58.63	10.76	4.57	1.72	20.48	2.32	0.56
10	cage13	1182.90	32.68	15.14	4.70	5.72	2.47	2.78	1.76
11	FEM_3D_thermal2	396.37	13.58	11.85	4.29	3.48	6.03	1.87	0.84
12	thermomech_dK	540.76	13.36	6.56	3.44	2.04	1.42	1.09	0.73
13	rgg_n_2_15_s0	87.68	2.14	4.06	3.11	0.60	0.89	0.37	0.30
14	rgg_n_2_16_s0	174.27	3.90	4.71	3.20	0.76	1.02	0.47	0.31
15	rgg_n_2_17_s0	342.79	7.93	5.83	3.25	1.16	1.50	0.73	0.37
16	rgg_n_2_18_s0	684.74	15.60	8.24	4.45	2.10	2.39	1.18	0.72
17	rgg_n_2_19_s0	1397.50	31.94	11.53	3.91	4.43	4.47	3.03	0.78
18	rgg_n_2_20_s0	2806.80	66.06	19.35	5.71	8.38	7.95	5.02	2.10
19	rgg_n_2_21_s0	5814.40	139.64	32.26	8.68	16.90	16.70	9.83	4.23
20	rgg_n_2_22_s0	11988.00	285.03	56.44	15.29	35.85	35.21	18.95	12.14
21	rgg_n_2_23_s0	24789.00	592.69	113.59	38.11	77.22	70.82	45.29	19.64
22	rgg_n_2_24_s0	52785.00	1221.10	230.46	75.73	167.91	140.41	103.40	45.07
23	qg.order100	51.98	15.65	98.24	12.99	35.51	3.94	21.61	10.66
24	twitch_gamers	525.55	51.39	143.25	75.60	2774.60	2.54	2112.30	1012.20
25	email_Enron	99.69	3.16	19.98	9.50	36.74	1.48	23.30	12.79
26	hollywood-2009	80926.00	579.20	1250.26	268.14	8526.83	69.42	7215.19	3263.75
27	indochina-2004	1266230.00	4105.95	2382.04	1450.90	55313.78	642.28	32072.10	14824.90
28	soc-LiveJournal1	7954.65	593.27	648.35	98.61	2083.13	119.15	1910.77	1003.30

Table 3.2 Average coloring time for each one of our implementations. Columns' headers have the meaning described in the itemization included in the main text. On the CPU-side, we indicate with GM_{s-imp} and GM_{a-std} our implementations of the GM synchronous and asynchronous algorithm, and with $JPL_{min-max}$ the JPL procedure. On the GPU side, we report the results of cuSparse, Gunrock, and Atos. The last two columns include our implementations on GPU. Once more, the graphs after the horizontal line (used as a separator) follow a power-law degree distribution.

Table 3.2 reports the coloring times (in milliseconds) for each graph, and the following algorithms:

- GM_{s-imp} , our improved synchronous version of Gebremedhin-Manne (Algorithm 11), running on the CPU.
- GM_{a-std} , our standard asynchronous version of Gebremedhin-Manne (Algorithm 10), running on the CPU.

- $JPL_{\text{min-max}}$, our implementation of the JPL min-max procedure (Algorithm 9) executing on CPU.
- cuSparse [76], i.e, the csrColor Cohen and Castonguay procedure (Section 3.2.6), running on the GPU.
- Gunrock [79], Gunrock’s algorithm (Algorithm 12), running on the GPU,
- Atos [80], a custom implementation of GM on GPU originally running on Volta architectures.
- JPL_{max} and $JPL_{\text{min-max}}$, our index-shift implementations (Algorithm 13) running on the GPU.

Notice that Table 3.2 considers only the coloring times and ignores all pre-processing and post-processing overheads, as done by the authors of all other approaches. We present results including pre- and post-processing times (comprising transfer times) in Table 3.3.

For the majority of the graphs, our two implementations of the Gebremedhin-Manne algorithm, running on parallel CPU, are slower compared to the implementations (both our own and state-of-the-art) executing on the GPU. The synchronous implementation is between 1 and 4 orders of magnitude slower than the best result we obtain on the GPU, while the asynchronous implementation is between 0 and 2 orders of magnitude slower. However, it is interesting how these slower implementations, especially the asynchronous one, perform comparably or even better on specific graphs, such as, twitch_gamers, email_Enron and qg.order100, hollywood-2009, indochina-2004, and soc-LiveJournal1, which for the most part are the ones to show a power-law degree distribution. The Cohen-Castonguay implementation, which on the other graphs shows the worst performance on the GPU, performs the best on these six graphs out of the GPU algorithms. To better understand this peculiar behavior, we analyze the topology of these graphs. All graphs share a large maximum degree: twitch_gamers has a maximum degree of 35279, email_Enron of 1383, hollywood-2009 of 11467, indochina-2004 of 256425, soc-LiveJournal1 of 20333, and all nodes of qg.order100 have a degree of 198. Furthermore, all graphs, except for qg.order100, have a power-law degree distribution, which implies the presence of a minimal number of nodes with an enormous number of edges. These degrees are very high when compared to the other benchmark graphs; among the others, the

highest degree is presented by ASIC_320ks, which has a maximum degree of 412, but an average degree of 5.7, meaning that the majority of its nodes have a much lower degree. On the implementations of the JPL algorithm for GPU, including the Gunrock implementation, nodes with these large degrees cause many issues of memory read instructions inside the loops on line 10 of Algorithm 13 and on line 14 of Algorithm 12, to fetch the random number associated with each neighbor. On graphs twitch_gamers and email_Enron, where node degrees vary, this also causes load imbalance, as threads assigned to color small-degree nodes are idle, while threads coloring large-degree nodes take a longer time to check all the neighbors. As the process of checking all neighbors is repeated at each iteration until the node with the most significant degree is colored, the total runtime of the algorithm becomes longer.

On the other hand, the Cohen-Castonguay algorithm does not suffer as much when running on these graphs because it computes the random values at runtime using hash functions, and the implementation assigns a larger number of colors for each iteration, ultimately completing coloring in fewer iterations. Similarly to the Gebremedhin-Manne CPU implementation, Atos shows excellent performance on these graphs. As Atos can assign a color reading the list of neighbors only once for each node, it potentially creates many conflicts; however, it also severely improves the overall performance when it converges faster. We can see this behavior both on the GPU and the CPU, except for the graph indochina-2004 in which the JPL CPU implementation outperforms the GM program. However, this is due to the poor thread scheduling that causes a huge oscillation in execution time over multiple runs, negatively impacting the average performances. GM-based methods still outperform JPL-based ones on this graph in the best case.

Table 3.3 compares wall-clock times considering the entire process, comprised of the pre-processing, coloring, and post-processing phases. The table illustrates the impact of our array generation process and the transfer time (to and from the GPU) on the overall execution time. The time required by our algorithm is divided into five steps such that the only time ignored are the ones required to read and write the graph to disk. The first three execution steps can be merged in Gunrock's pre-process time. Even though the transfer time is usually more significant than the execution time, we can still show that our implementation is able to solve all of our instances in less time than Gunrock. Speedups vary from 2.17x to 61.07x with a geomean speedup of 7.43x. Moreover, please notice that, to the best of

#	Graph	Gunrock				Total	JPL _{min-max}				Speedup	
		Preprocess	Process	Postprocess	Total		Randomization	Preprocess Allocation	Transfer	Process		Postprocess
1	af_shell3	74.74	28.09	6.04	108.87	2.91	0.34	6.44	5.50	0.24	15.43	7.06
2	apache2	57.60	1.19	6.54	65.33	4.10	0.26	2.23	0.39	0.32	7.30	8.95
3	ecology2	51.02	0.87	8.95	60.84	5.75	0.26	2.51	0.36	0.41	9.29	6.55
4	G3_circuit	58.78	1.18	13.93	73.89	9.01	0.31	3.95	0.57	0.61	14.45	5.11
5	offshore	53.98	3.29	2.95	60.22	1.50	0.28	1.76	0.82	0.17	4.53	13.29
6	parabolic_fem	50.16	1.06	4.99	56.20	2.96	0.27	1.66	0.38	0.26	5.53	10.16
7	thermal2	52.59	2.03	10.77	65.39	7.02	0.32	4.78	0.74	0.59	13.45	4.86
8	ASIC_320ks	58.98	3.42	3.11	65.50	1.81	0.27	0.94	1.29	0.20	4.51	14.53
9	atmosphodl	53.23	1.71	11.76	66.71	7.31	0.31	4.00	0.64	0.51	12.77	5.22
10	cage13	53.14	5.72	5.20	64.07	2.48	0.25	2.94	1.60	0.24	7.51	8.53
11	FEM_3D_thermal2	49.73	3.48	2.02	55.23	0.86	0.27	1.32	0.95	0.10	3.50	15.78
12	thermomech_dk	54.49	2.04	2.32	58.85	1.17	0.27	1.19	0.51	0.14	3.28	17.94
13	rgg_n_2_15_s0	53.26	0.60	0.49	54.35	0.20	0.27	0.20	0.19	0.03	0.89	61.07
14	rgg_n_2_16_s0	61.82	0.76	0.76	63.34	0.39	0.27	0.35	0.24	0.05	1.30	48.72
15	rgg_n_2_17_s0	48.31	1.16	1.27	50.74	0.82	0.28	0.66	0.37	0.09	2.22	22.86
16	rgg_n_2_18_s0	61.40	2.10	2.71	66.21	1.59	0.28	1.39	0.63	0.17	4.06	16.31
17	rgg_n_2_19_s0	51.60	4.43	5.04	61.08	3.18	0.28	2.88	1.11	0.26	7.71	7.92
18	rgg_n_2_20_s0	69.09	8.38	10.06	87.53	6.38	0.49	6.25	2.24	0.45	15.81	5.54
19	rgg_n_2_21_s0	71.55	16.90	18.47	106.92	12.70	0.40	11.64	4.69	0.79	30.22	3.54
20	rgg_n_2_22_s0	105.89	35.85	32.93	174.66	23.34	0.61	24.03	9.79	1.52	59.29	2.95
21	rgg_n_2_23_s0	149.53	77.22	62.62	289.37	38.21	0.94	49.69	22.38	2.88	114.10	2.53
22	rgg_n_2_24_s0	246.17	167.91	122.67	536.76	60.34	1.59	81.43	52.24	6.38	201.98	2.65
23	qg_order100	49.59	35.51	0.64	85.75	0.06	0.27	0.74	9.01	0.01	10.09	8.50
24	twitch_gamers	66.67	2774.63	3.77	2845.08	1.00	0.32	4.97	1012.45	0.11	1018.85	2.79
25	email-Enron	62.16	36.74	0.49	99.39	0.25	0.26	0.18	14.56	0.03	15.28	6.50
26	hollywood-2009	141.02	8086.35	12.41	8239.78	0.26	0.75	38.61	3510.13	0.47	3550.21	2.32
27	indochina-2004	273.73	58306.92	57.27	58637.92	39.28	1.70	104.76	15933.30	2.52	16081.56	3.65
28	soc-LiveJournal1	119.12	2230.96	39.58	2389.66	1.08	0.68	30.56	1066.01	1.67	1100.00	2.17

Table 3.3 Detailed comparison of our JPL min-max approach against Gunrock. Notice that Gunrock pre-processing times are roughly equivalent to the sum of our times, including the vector randomization, the GPU allocation, and the CPU-to-GPU transfer time. The post-processing phase includes the GPU-to-CPU transfer time.

our knowledge, the transfer time is rarely considered in literature when comparing CPU and GPU performances. After all, it is more an architectural issue than an algorithmic one, and it can currently be reduced by compression and decompression strategies, the Zero-Copy memory approach, or by reading the data directly from disk, which are methodologies available on the newest NVIDIA cards. Furthermore, in an environment in which CPUs and GPUs collaborate to solve a set of problems, it is unclear whether the memory transfer time has to be “added” to the GPU and not to the CPU performances.

Notice that we show the transfer times for our $JPL_{\min\text{-max}}$ approach, but at the same time, our measurements are also valid for JPL_{\max} . The only difference between the two is the coloring time. Moreover, we put effort into optimizing the transfer times using CUDA Streams. This optimization allowed us to reduce the memory transfer costs and avoid useless synchronizations that proved to be small bottlenecks. However, as we used CUDA streams to gather precise timings, we performed GPU-CPU synchronizations after each phase. Furthermore, for this comparison, we add all of the times together to compute the speedup; operations like the randomization of the weights vector that is performed on the GPU using the *fast_rand* function detailed before can be performed while reading the graph itself from the disk since the number of nodes is known from the beginning of the process.

Figure 3.6 plots the speedups of all GPU implementations over the Gunrock implementation. We evaluate the ratio between the computation time of the Gunrock strategy and all other methods X, i.e., $t(\text{Gunrock})/t(X)$, and displayed these values on a logarithmic scale on the y-axis.

Obtaining the minimum and the maximum speedups with our JPL implementations on the same graphs is not coincidental. The two implementations are coded such that $JPL_{\min\text{-max}}$ should color twice as many nodes as JPL_{\max} ; thus, it is not surprising that the processing stage is twice as fast on the same graph structure. In Figure 3.7, we display the speedup obtained by coloring two independent sets per iteration ($JPL_{\min\text{-max}}$) over the standard approach of coloring a single one (JPL_{\max}).

To better understand the differences between our $JPL_{\min\text{-max}}$ implementation and the state-of-the-art implementation from the Gunrock library, we use the Nsight Compute profiler to collect information on their runtimes. Nsight Compute collects data on every kernel launched during the execution. Some of the metrics collected

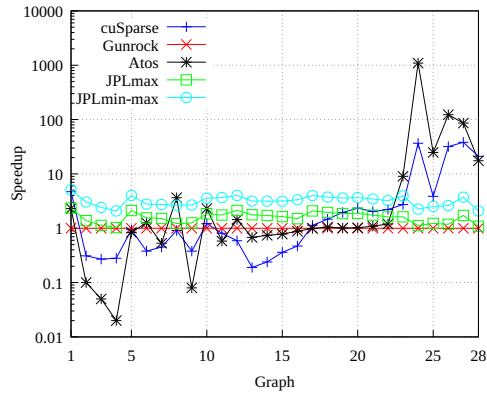


Fig. 3.6 Speedups of our implementations JPL_{\max} and $JPL_{\min-\max}$ against CuSparse, Gunrock, and Atos. The Gunrock procedure (in red color) is used as a reference and normalized to one.

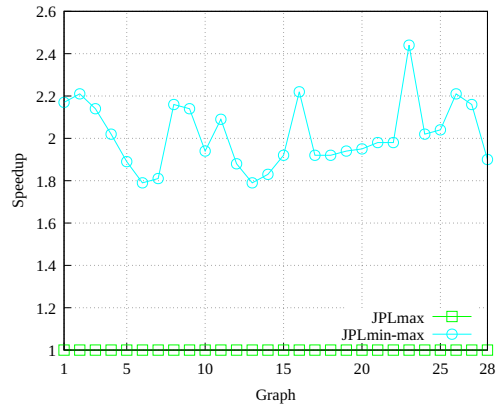


Fig. 3.7 Speedup of our $JPL_{\min-\max}$ approach (dealing with two independent sets for each iteration) over our JPL_{\max} methodology (dealing with a single independent set). The expected 2X factor is reached on average as overheads are negligible.

include grid dimensions, execution time, the average number of threads active per warp (to estimate divergence), cache hit rate, and many more. From the profiles, we know that both implementations rely on more than one kernel to perform the coloring. For $JPL_{\min\text{-max}}$, the main kernel that actively performs the coloring (*color_jpl_kernel*) is followed by two auxiliary kernel calls (*DeviceReduceKernel* and *DeviceReduceSingleTileKernel*) used to compute the number of uncolored nodes after every iteration. For the Gunrock implementation, the coloring is performed by the kernel named *Kernel*, and auxiliary tasks are performed by the kernels named *GetEdgeCounts*, *launch_box_cta_k*, and *gen_sequenced*. Among these kernels, *gen_sequenced* is called once every two calls of *Kernel*, and it is the kernel that regenerates the array of random numbers. Since the execution time of the auxiliary kernels in both algorithms is negligible compared to the execution time of the main kernels, we do not consider them, as they have a limited impact on the overall execution time. In Figure 3.8, we represent the execution times of the main kernels (*color_jpl_kernel* for $JPL_{\min\text{-max}}$ and *Kernel* for Gunrock/color) during the coloring of the graph *af_shell3*.

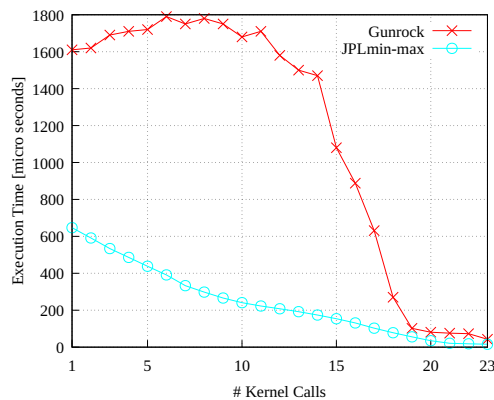


Fig. 3.8 Elapsed time to complete each kernel launch within our $JPL_{\min\text{-max}}$ strategy and the one delivered by Gunrock.

Figure 3.8 shows that each new kernel launched by the $JPL_{\min\text{-max}}$ implementation terminates its execution slightly faster than the previous kernel. The first kernel achieves the maximum execution time, taking 647 μs , while the faster kernel is the last one, terminating its execution in 16 μs . The execution times of the kernels run by the Gunrock/color implementation can be divided into two phases. In the first phase,

encompassing the first 14 kernel launches, the execution times oscillate around the value of 1669 μs , with a maximum of 1790 μs for the 6th kernel, and a minimum of 1470 μs for the 14th kernel. The second phase, spanning from the 15th kernel launch up to the last one, approximately follows a negative exponential trend, going from a maximum of 1080 μs for the 15th kernel to a minimum of 43 μs for the 23rd and last kernel launch. Some caching issue likely causes the discrepancy in the two phases of the Gunrock implementation. Indeed, the kernel is written such that the random value associated with the current node is not cached, and needs to be read multiple times inside the loop on lines 15 and 18 of Algorithm 12. Multiple reading operations cause extremely high execution times for the first iterations, which rapidly drop in the second phase, after the majority of the nodes have been colored. On the other hand, our $\text{JPL}_{\text{min-max}}$ implementation does not suffer from this problem, as the random values are cached in registers on lines 9 and 11 of Algorithm 13. Our analysis of kernel running times also includes the version of our $\text{JPL}_{\text{min-max}}$ that does not implement index shifting. We do not report those runtimes in Figure 3.8 as they are very similar and follow the same trend as the ones reported for strategy $\text{JPL}_{\text{min-max}}$. The version without index shifting is on average 13% faster on the first 16 kernels, and around 175% slower on the remaining 7 kernel calls. However, the version without index shifting terminates only after 32 kernel runs, meaning that more colors are used in the solution.

Table 3.4 reports the average number of colors (over 10 runs and rounded to the nearest integer) used by our implementations over all our test graphs. The data shows how the two CPU implementations of the Gebremedhin-Manne algorithm (namely, $\text{GM}_{\text{s-imp}}$ and $\text{GM}_{\text{a-std}}$) consistently generate solutions using fewer colors than the GPU implementations. As the Atos approach is based on GM, this is also true for Atos. Between the two GM implementations, the improved version uses fewer colors in all graphs other than *apache2* and *ecology2*. This behavior is expected as the improved algorithm is formulated to reduce the number of colors generated during the first coloring step of the algorithm. However, since the coloring is performed non-deterministically, the final improved solution is not guaranteed to use fewer colors than the standard solution. Figure 3.9 uses the synchronous improved GM implementation as a baseline to compare the number of colors of all other methods. The figure reports the data of Table 3.4 as a percentage increase, computed as $((c(I) - c(\text{GM}_{\text{imp}})) / c(I))$. Averages for all implementations are displayed as dotted lines with the same colors.

#	Graph	CPU-based			GPU-based				
		GM _{s-imp}	GM _{a-std}	JPL _{min-max}	cuSparse	Gunrock	GM _{a-std}	JPL _{max}	JPL _{min-max}
1	af_shell3	26	27	46	80	49	28	47	46
2	apache2	5	4	13	33	16	7	12	12
3	ecology2	5	4	10	32	12	4	10	10
4	G3_circuit	5	5	10	32	11	5	10	10
5	offshore	11	13	23	48	27	14	24	24
6	parabolic_fem	6	6	12	32	13	6	12	12
7	thermal2	7	7	12	33	15	8	12	12
8	ASIC_320ks	6	8	15	48	18	9	17	15
9	atmosmodd	5	6	13	35	14	6	13	13
10	cage13	14	16	37	64	41	18	37	37
11	FEM_3D_thermal2	15	18	36	64	38	18	36	36
12	thermomech_dK	12	13	20	48	21	14	20	20
13	rgg_n_2_15_s0	13	14	21	48	20	15	21	21
14	rgg_n_2_16_s0	15	17	23	48	23	17	23	23
15	rgg_n_2_17_s0	15	16	24	48	26	17	24	24
16	rgg_n_2_18_s0	17	19	25	50	27	18	25	25
17	rgg_n_2_19_s0	18	19	27	48	29	19	27	27
18	rgg_n_2_20_s0	18	19	29	57	33	20	30	29
19	rgg_n_2_21_s0	19	20	30	59	32	22	30	30
20	rgg_n_2_22_s0	20	22	31	64	33	23	32	31
21	rgg_n_2_23_s0	22	24	33	64	34	25	34	33
22	rgg_n_2_24_s0	23	23	35	64	37	26	35	35
23	qg.order100	121	147	238	239	221	142	246	237
24	twitch_gamers	112	118	468	504	509	132	469	469
25	email_Enron	36	42	118	146	127	45	120	117
26	hollywood-2009	2209	2209	2287	2272	2274	2209	2299	2272
27	indochina-2004	6849	6849	6985	6983	6986	6849	6983	6982
28	soc-LiveJournal1	347	340	513	538	518	341	513	513

Table 3.4 The average number of colors for each one of our implementations. On the CPU side, we present our implementations of the GM synchronous and asynchronous algorithm (GM_{s-imp} and GM_{a-std}, respectively), and the JPL procedure (JPL_{min-max}). On the GPU side, we report the results of cuSparse, i.e., the csrColor Cohen and Castonguay implementation, Gunrock, i.e., the Gunrock’s algorithm, and Atos from Chen et al. The last two columns include our implementations on GPU.

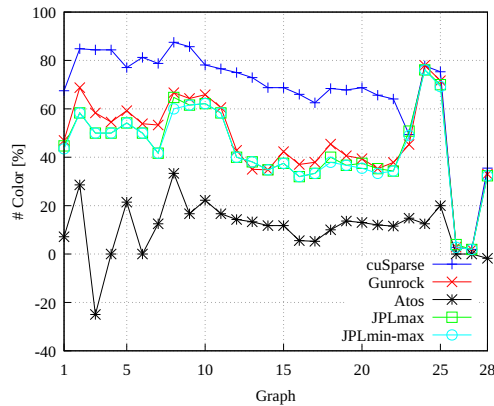


Fig. 3.9 Percentage variations in the number of colors used by the different GPU-based methods with respect to GM_{S-imp} used as a reference and CPU-based.

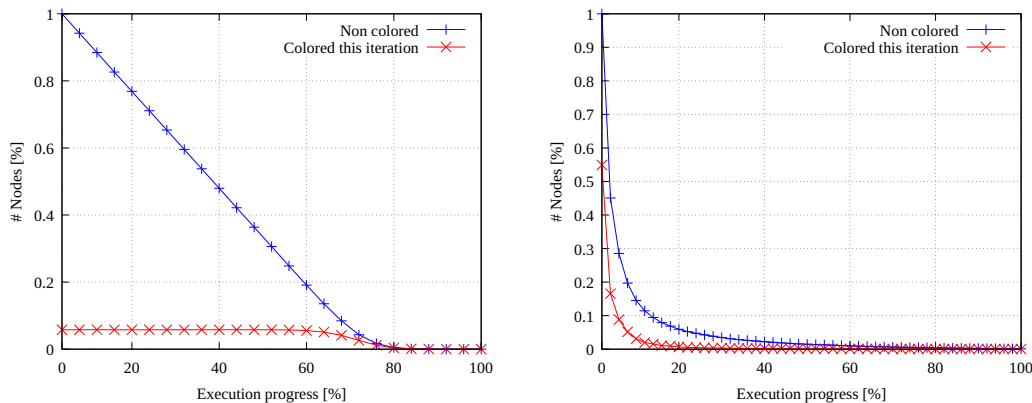
On all graphs other than *qg.order100*, the state-of-the-art implementation of the Cohen-Castonguay algorithm is the one to generate the most colors, with an average percentage over GM_{S-imp} of 55.5%. The three other implementations manage to use fewer colors. Our $JPL_{min-max}$ implementation uses 7% fewer colors on average than the state-of-the-art implementation of the same algorithm on Gunrock. Our JPL_{max} implementation shows mixed results, using more colors than the Gunrock implementation in some graphs and matching the colors of $JPL_{min-max}$ on other graphs, including the *rgg* graph family. JPL_{max} reports an average percentage over GM_{S-imp} of 46.17%, Gunrock of 39.24%, and 36.39% for $JPL_{min-max}$.

3.2.10 Performance analysis

To study how our algorithms face load imbalance and variable-size frontier sets, we present the following analysis.

Figure 3.10 represents the percentage of the nodes colored at each iteration, and the ones which remain uncolored, as a function of the number of main coloring iterations. We report these values for the graph *rgg_n_2_24_s0*, but this behavior is typical to all graphs on which our algorithm performs optimally. The number of nodes colored during each step remains practically constant, and it drops only when the execution is ending, as the remaining uncolored nodes are only a tiny fraction

of the originals but they still require multiple passes to be colored. This behavior is characteristic of an efficient solution and indicates that our algorithm works at its best and is usually much faster than all competitors.



(a) A balanced behavior on a graph including vertices with low or average degree. (b) An unbalanced behavior on a graph including a few vertices with a very high degree.

Fig. 3.10 The y-axis represents a percentage of the nodes, whereas the x-axis represents the execution progress.

On the contrary, Figure 3.10b represents a graph with power-law degree distribution, more specifically, *indochina-2004*. However, as for the previous analysis, this behavior is ubiquitous for all graphs of this type. This graph has a conformation for which most nodes can be colored in very few passes, as they have very few neighbors and are located toward the the graph's edges. The remaining nodes are those in the most populated areas and require numerous iterations to be successfully colored. Although it is possible to imagine a parallelism between the number of uncolored nodes and the size of the search frontier, this is not appropriate for the JPL approach, which, at each iteration, checks all nodes by skipping those that have already been colored. This factor implies that the size of the frontier is constant throughout the execution. As a consequence, the number of threads that need to operate after a node is received decreases as the number of iterations advances. This consideration, in turn, increases the load imbalance and the divergence. In particular, this is true for graphs that follow the power law. The NVIDIA Nsight Compute shows 20 active threads per warp on average for *rgg_n_2_24_s0* but only 12 for *indochina-2004* (and with a smaller number of instructions issued per cycle). Even the warp occupancy shows an imbalanced workload, as it is equal to 90% in the first case and 24% in the

second one. These values show that the higher the number of steps required by the JPL procedure, the more expensive the operation becomes. On the other hand, Atos is more efficient for this graph structure, as it has over a 72% warp occupancy on average on *indochina-2004*, showing better use of the threads on the GPU.

3.2.11 Conclusions

This section describes, studies, and implements the most efficient state-of-the-art graph coloring algorithms running either on multi-core CPUs or many-core GPUs using CUDA. We put particular attention to the JPL algorithm, which improves the algorithm efficiency by coloring independent sets of vertices.

We present two GPU implementations of this algorithm, which differ in the number of independent sets colored at each iteration. We enhanced these implementations with “value permutation”, a method to generate a random permutation of a set of unique items, and “index shifting”, a technique to simulate a circular array shift with a meager cost compared to the original strategies. These techniques improve the runtime of the algorithms, and they also reduce the number of colors used for coloring a graph.

We compared our implementations with three state-of-the-art implementations of graph coloring, namely, NVIDIA’s *cuSparse*, *Gunrock*, and *Atos*. As far as the pure coloring procedure is concerned (without pre- and post-processing), we show that our fastest implementation presents geomean (harmean) speedups of 3.16x (3.05x) against *Gunrock*, 4.09x (3.06x) against *cuSparse*, and 4.45x (2.21x) against *Atos* on mesh-like graphs. When we concentrate on the entire process (pre-processing, processing, and post-processing phases, including transfer times) our implementation has geomean (peak) speedups of 7.43x (61.07x) against *Gunrock* (the fastest of the competitors). At the contrary, the algorithm performs significantly worse when applied to scale-free graphs, where it is competitive only against *Gunrock*, the other implementation of the JPL algorithm. It shows a geometric mean (harmonic mean) of 2.76x (2.71x) against *Gunrock*, 0.13x (0.11x) against *cuSparse*, and 0.03x (0.01x) against *Atos*. At the same time, our approach can generate solutions using less colors than the other JPL-based procedures. With graphs that contain vertices with a huge number of arcs, our procedure (as all other JP-based algorithms) is slower than GM-based procedures and *Atos*. Since computing the characteristics of a given

graph is a task that can be performed while reading or storing it, it is consequently possible to use a multi-engine approach and select the best algorithm to solve each instance as quickly as possible.

Further research is needed to study how our implementation can be further improved. Indeed, it is interesting to notice how our index shift technique stemmed from our initial decision to use the value permutation strategy to obtain unique numbers. Gathering random numbers from a uniform distribution would ultimately incur in a too high overhead to be recomputed. For this reason, we encourage further research to change the variables at play in an algorithm. Moreover, it would also be beneficial to implement other algorithms on many-core GPU architectures, as the speedup provided by those devices is substantial but graph algorithms rely on a lot on memory operations and researchers have been unable to exploit their power completely with graphs.

Moreover, we started developing a Vulkan-based version of the algorithm. This uses the Vulkan Kompute [12, 13] library, which allows for GPU computations but, unlike CUDA, is available on multiple operating systems and GPU vendors, such as NVidia, AMD and Intel. However, this is still under research and is left as a future work, as results are still mixed on our NVidia RTX 3070.

3.3 McSplit+PR

In this section I will talk about the work done in [66]. This is an improvement over the stat-of-the-art algorithm McSplit and its variants, as we will see in the next sections.

3.3.1 Introduction

Graphs are flexible structures that allow us to model many elements of human knowledge through a mathematical abstraction. In particular, graphs can be very good representations of relationships between objects. Graphs find many applications in fields such as chemistry [88], social networks [89], web searches [90], security threat detection [91], modeling dependencies between different software components [92], hardware testing and functional test programs [93].

In the rest of this chapter I will describe how we improved the computation of the Maximum Common Subgraph (MCS) between two graphs by improving the McSplit Algorithm. Even if the problem has been appearing in the scientific literature since the 70s [94, 95], one of the most efficient state-of-the-art algorithm for finding MCS is McSplit, introduced in 2017 by McCreesh et al. [96]. McSplit is a branch-and-bound algorithm that recursively computes new solutions by pairing vertices selected from the two graphs. The core idea is to label all vertices based on the connection they have with already selected nodes. After that, the algorithm efficiently prunes the search tree taking into account those labels and a formula computing the upper bound for the size of the current solution. The approach is quite efficient in maintaining low memory profiles and pruning the search space. Unfortunately, it considers all possible vertex pairs, one vertex from the first and one from the second graph, and its performances strongly depend on the vertex sorting heuristic. The original version of McSplit statically sorts the vertices of both graphs based on their degree. This order is then maintained unaltered for the entire process, and it is the most impairing element of the procedure. Many vertices may have identical degrees, making it impossible to discriminate between them. Moreover, there is no way to prioritize a promising pair discovered during the execution of the algorithm. In our approach, we exploit the core of the original McSplit procedure, but we replace the static sorting heuristic with sharper ordering techniques.

McSplitRL [97], McSplitLL [98], and McSplitDAL [99] already brought an improvement over the original sorting heuristic of McSplit. McSplitRL uses a Reinforcement Learning approach to refine the order of the vertex selection. McSplitLL, based on McSplitRL, outperforms its predecessor by using a technique called Long Short Memory which deals with nodes with specific characteristics. McSplitDAL builds upon McSplitLL, introducing a technique called Dynamic Action Learning, which improves the reward function of McSplitRL. However, these techniques use the original McSplit sorting heuristic as a tie-breaker when selecting vertices.

In this work, we present a new vertex selection heuristic that is able to improve the performances of McSplit, McSplitLL, and McSplitDAL. In particular, we propose to use PageRank [90], the former algorithm behind the Google search engine, as a vertex selection heuristic, exploiting its capabilities to work on both directed and undirected graphs. We use PageRank both as a standalone or as a tie-breaking heuristic, using it to classify vertices and then combining it with other techniques such as McSplitLL or McSplitDAL.

In our experimental analysis, we compare our algorithm with McSplit and its variants. We tested 400 graph pairs, selecting the graphs from the largest publicly available graphs at [100] and choosing at least one graph pair for each graph category. We set the timeout for each experiment to 60 seconds to quickly grab the convergence speed of each algorithm. Overall, we can improve McSplit, McSplitRL, McSplitLL, and McSplitDAL in up to 77% of the graph pairs considered. Moreover, we obtain an improvement in terms of the final size of the solution subgraph up to 7%.

3.3.2 Background

In addition to what we saw in Section 4.1, I will introduce some concepts useful for understanding of this work.

We say that H is a *subgraph* of G if

$$V(H) \subset V(G) \wedge E(H) \subset E(G)$$

that is, the vertices and edges of H are a subset of the vertices and edges of G . A graph H is an *induced subgraph* of G if H is a subgraph of G and contains all the edges between its vertices of the original graph G .

Graph isomorphism is the problem of detecting if there is a bijection between two graphs G and H such that

$$\forall v_1, v_2 \in H \in E(H) \iff \{v_1, v_2\} \in E(G)$$

that is, if two graphs have the same structure. Verifying whether two graphs are isomorphic is known to be NP [101], even if the exact complexity inside that class is unknown.

A subgraph is a subset of a graph's vertices (or nodes) and edges (or links). The terms vertex and node will be used interchangeably in this paper.

The Maximum Common Subgraph (MCS) problem between graphs G and H , requires finding the most extensive graph simultaneously isomorphic to a subgraph of G and H . In particular, the Maximum Common Induced Subgraph (MCIS) focuses on finding the induced subgraph with all the vertices in common between two graphs. The problem is known to be NP-complete [102].

In our case, we focus on undirected, unlabeled, and unweighted graphs, as they represent the worst case scenario for the Maximum Common Subgraph computation.

3.3.3 McSplit

McSplit [96] is a branch-and-bound recursive algorithm for finding the MCS between two graphs.

The authors define a *label class* as a set of vertex pairs (belonging to the first and the second graph) having the same connections toward the vertices belonging to the current solution. As McSplit uses labels to find possible couplings between vertices, the original algorithm also provides a way to create those labels based on the adjacency lists of the vertices.

Algorithm 14 The simplified version of the original McSplit algorithm.

```

BEST ← ∅
MCS (G, H, M)
  if |M| > |BEST| then
    BEST ← M
  end if
  if CalculateBound() < |BEST| then
    return
  end if
  label_class ← SelectLabelClass(G, H)
  G' ← G
  while G' ≠ ∅ do
    v ← SelectVertex(G, label_class)
    G' ← G' \ {v}
    for w ∈ getVertices(H, label_class) do
      M' ← M ∪ (v, w)
      H' ← H \ {w}
      G' ← UpdateLabels(G', v)
      H' ← UpdateLabels(H', w)
      mcs(G', H', M')
    end for
  end while
  mcs(G', H, M)

```

Algorithm 14 provides a simplified version of the McSplit algorithm. It takes as inputs the two graphs, G and H , as well as the current solution M . Label classes

are used to guide the algorithm in finding the solution to the problem. The label class is a classification of each couple of vertices belonging to (G, H) . First, the algorithm assigns the current solution to the best one (line 3), in case the current solution has a larger size (line 2). Notice that the best solution *BEST* is initially empty (line 1). Then, the algorithm calculates the upper bound B for the current path (line 5). If this upper bound is less than the size of the best solution, the current solution cannot be improved along the current path; thus, the algorithm backtracks (line 6). Otherwise, the algorithm keeps improving the current solution. The bound is computed as shown in Equation 3.1.

$$B = |M| + \sum_{l \in L} \min(|\{v \in G \setminus M : L(v) = l\}|, |\{w \in H \setminus M : L(w) = l\}|) \quad (3.1)$$

When improving the current solution, McSplit tries to build a larger solution by virtually removing a couple of vertices with the same label from the respective graphs, updating the labels (lines 16-17) and trying to explore recursively all possibilities starting from the current solution (line 18). In each iteration of the algorithm, the selection of a vertex pair occurs in three distinct stages. Firstly, the most promising label class is identified, followed by selecting a vertex from the set of vertices belonging to that label class in the graph G (line 12). Subsequently, all vertices $w \in H$ of the chosen label class are gathered (line 13), and then individually selected one by one (line 15). Once $v \in G$ is selected, the current *mcs* instance uses recursion to explore all solutions that include v and all the nodes of the received partial solution M , therefore at line 21 an additional recursive call is introduced to explore all the other solutions that include M but exclude v . Ultimately, as every vertex couple has been explored (line 10), the procedure returns the best solution.

To explore all possible vertex pairs, McSplit uses two different heuristics. The first one is used to select the next label class. The second one is adopted to choose the next vertex to add to the final graph. The former (line 8) chooses the label class with the smallest maximum size between G and H , i.e., $\max(|G|, |H|)$. The latter, instead, prioritizes vertices in G with the most significant degree, where the degree is the number of links (inward and outward) of the vertex. In particular, for selecting the next vertex (line 11), McSplit heuristically considers the degree of the vertex, choosing each time the vertex with the most considerable degree and removing it from the graph. We will refer to this approach as the *Node Degree*, or simply the *Degree* heuristic.

3.3.4 McSplit variants

Many notable variants of McSplit have been developed to improve over the original algorithm. This section briefly describes some of the most noticeable and recent ones.

McSplitSD

McSplit works asymmetrically on the two graphs since it selects a vertex from G and then searches for a matching vertex in H . This approach may unbalance the algorithm, making it perform better or worse, depending on the characteristics of the first graph. Among other strategies, Trimble [103] proposes McSplitSD, which sets as the first graph the denser one of the pair. The density K of a graph is evaluated through Equation 3.2, using the number of edges and vertices of the two graphs to express the *density extremeness*:

$$K(G) = \frac{|E(G)|}{|V(G)| \cdot (|V(G)| - 1)} \quad (3.2)$$

The two graphs G and H are swapped when the inequality

$$\left| \frac{1}{2} - K(G) \right| > \left| \frac{1}{2} - K(H) \right|$$

is true.

McSplitRL

Liu et al. [97] proposes McSplitRL, a novel approach that extends the standard McSplit using Reinforcement Learning. This approach keeps two vectors, one for the vertices of G and the other for the vertices of H , which contain the rewards of each node. Therefore, the node selection heuristic is based on finding the node with the highest reward. The authors devised a scoring system for a given action using Equation 3.3:

$$R(v, w) = \frac{\sum_{(V_l, V_r) \in E_v} \min(|V_l|, |V_r|) - \sum_{(V'_l, V'_r) \in E_{v'}} \min(|V'_l|, |V'_r|)}{\quad} \quad (3.3)$$

Given a set of label classes of the initial graphs at a given point of the search, E_v , and the subsequent set of label classes, E'_v , generated by including a new couple of

vertices to the current solution, Equation 3.3 calculates the reduction of the size of the label classes. The size of a label class is considered as the minimum of $|V_l|$ and $|V_r|$, which are the number of vertices belonging to the label class respectively from the first or the second graph. Thus, this method can be seen as a bound reduction and tends to prefer nodes whose resulting branching cause a higher reduction of the bound, thus cutting as many branches as possible in subsequent steps of the algorithm.

McSplitLL

Zhou et al. [98], starting from McSplitRL, build a more sophisticated version of the tool called McSplitLL. Their solution introduces a new heuristic called Long Short Memory (LSM) and a method to be used in a specific situation called Leaf Vertex Union Match (LUM). The new heuristic uses Equation 3.3 but stores the rewards in a vector for nodes of G and a matrix for the nodes of H , allowing to reward each possible node pair separately $(v, w) \in (G, H)$.

However, since rewards may become huge, an asymmetric decay is used, following a long-short-term approach, which halves both G and H rewards when their respective thresholds are exceeded. Rewards for single nodes v decay faster than the rewards for pair of nodes (v, w) ; thus, node pairs have a smaller threshold.

Moreover, the LUM heuristic introduces a more optimized strategy to handle leaf nodes. A node is considered a leaf if it is adjacent to only one vertex of a given graph, and it has been proved it can always be added to the current subgraph if its only neighbor is part of it as well. Thus, whenever a leaf from the left graph and a leaf from the right graph is found, the pair formed by these two nodes is added to the current solution.

McSplitDAL

Liu et al. introduced McSplitDAL [99]. This algorithm is the most recent version of McSplit, and it is built upon McSplitRL and McSplitLL. This algorithm mainly introduces two new ideas. A new value function called Domain Action Learning (DAL) and a hybrid learning policy for choosing the next vertex to match. The DAL value function aims to take into account, when branching, not only the reduction of

the upper bound but also the simplification of the problem occurring after the branch. This feature can be implemented by adding an additional term to the reward defined in Equation 3.3, granting a higher reward to the vertices whose generated partitions have a higher cardinality, when these vertices are added to the solution:

$$R(v, w) = \sum_{(V_l, V_r) \in E_v} \min(|V_l|, |V_r|) - \sum_{(V'_l, V'_r) \in E_{v'}} \min(|V'_l|, |V'_r|) + |E_{v'}| \quad (3.4)$$

Moreover, the hybrid branching policy of this approach has the primary goal of overcoming a possible “Matthew effect”, which causes the algorithm to continue branching on a subset of nodes with very high rewards getting trapped in a local optimum. The authors believe this can be overcome by switching from the RL to the DAL policy (and vice versa) after a fixed number of iterations without improvement, allowing to dynamically change the strategy for selecting nodes.

For brevity, in this paper, we use the term *McSplitX* to generically identify the original McSplit or one of its variants, i.e., McSplitLL, or McSplitDAL.

3.3.5 Other Approaches

Many algorithms have been presented to solve the MCS problem, using strategies that differ from the original McSplit. Among those, we would like to mention the following. Levi [104] casts the MCS problem onto the Maximum Common Clique problem. McCreesh et al. [105] and Vismara et al. [106] follow the previous approach while exploiting constraint programming to solve the problem. Other approaches take a step back, adopting parallel computation capabilities of General-Purpose computing on Graphics Processing Unit (GPGPU) [107], to enhance McSplit on modern devices. A set of heuristics to tackle the MCS problem with more than two graphs has been developed by Cardone et al. [108]. However, the most promising heuristics work by analyzing graphs in couples and later merging the results, thus still motivating the research on MCS techniques working on pairs of graphs.

3.3.6 The PageRank Algorithm

PageRank [90] is an algorithm developed by Google that, given a network of web pages, generates the probability of reaching a page through a finite sequence of random clicks. PageRank was the algorithm used by Google to sort the results of its web engine searches. However, it is not used anymore, as its patent expired in 2019.

PageRank is usually implemented on a generic graph, so to account for different web pages, it considers directed and unweighted graphs. A link from one web page takes the user to another web page, but the way back is not guaranteed. However, we can also use it on undirected graphs, as we can think of them as directed graphs with both forward and backward edges between each node pair.

Algorithm 15 implements our PageRank algorithm, and it is strongly inspired by a public version². In Algorithm 15, we use the notation $adj(G)$ to refer to the indices of the adjacency matrix of graph G .

The Damping Factor (DF), initialized in line 1, represented a person's probability of stopping clicking random links. We decided to follow Brin et al. [90] recommendation for the value of the DF , and we set its value at 0.85. In line 2, we set the acceptable error ε at an arbitrary value. Experimentally, we discover that the smaller the epsilon (i.e., the more we increase the precision of the procedure), the better the results, as the rankings tend to be more diverse. However, as the original algorithm accepts integers numbers, we also want to be able to map integers to ranks; thus, we chose for ε a precise enough number that would surely not overflow any 32-bit integer.

PageRank can be described as a Markov chain. Thus, we build a stochastic matrix representing the graph in line 13, based on the previously computed links going out from each node in line 12. Computing the outgoing links is trivial and is not shown in the algorithm. On the contrary, the computation of the stochastic matrix is represented in function `StochasticGraph`, from line ?? to line 11. Assuming that each node has a unitary amount of information flowing outwards to the neighbors, the matrix identifies how much of that information is flowing through each of the adjacent edges. In line 14 we transpose the stochastic matrix, and outgoing links are replaced with incoming links and vice versa. PageRank ranks nodes based on their incoming links; thus, the inversion is necessary for the generality of the algorithm.

²<https://github.com/purtroppo/PageRank>

For undirected graphs, this might represent an unnecessary step; however, as McSplit works on directed and undirected graphs, this must be true also for its intermediate stages. On line 16, we pre-allocate the results of the previous iteration and set them to zero.

In line 18 we calculate the ratio between the incoming or outgoing links and the size of the graph. The core section of the evaluation is included from line 21 to line 34. First, we zero the results for the current iteration. Then, we compute the current rank by adjusting the previous results, approximating at each iteration the clicking probability, and discounting them by the DF . On line 31, we update the error on the measurement, and on line 33 we update the result vector p . The algorithm terminates when ($error < \epsilon$) in line 21; this condition is triggered when the rankings converge, reaching a stable configuration.

As we consider it trivial, we do not show the float to integer conversion in Algorithm 15.

3.3.7 The main idea behind our approach

The main target of this work is to improve the vertex selection heuristic. In particular, we are interested in heuristics that can classify the vertices of the two graphs. From our perspective, a good heuristic should follow the guidelines presented by Marti et al. [109]:

- The solution should be nearly optimal.
- The heuristic should require low computational effort.

In our heuristics, we also aim to generate classifications as diverse as possible for ranking the vertices. Moreover, we would like heuristics to classify a vertex with a single number instead of representing it as a vector. Although vectors have already been used in MCS solutions, due to the nature of the problem, using a mathematical vector incurs possible downfalls. More specifically, vectors may require more computational power to retrieve a classification than using single integers and the results may depend on the lexicographical order of the vertices. With these considerations in mind, we focus on a classification of vertices based on single numbers. In particular, we developed different heuristics for classifying vertices:

Algorithm 15 Our version of the popular PageRank algorithm, implemented on an adjacency matrix representing the graph G .

```

 $DF \leftarrow 0.85$ 
 $\varepsilon \leftarrow 0.00001$ 
STOCHASTICGRAPH ( $G, out\_links$ )
 $G_s \leftarrow [0.0] * |G|$ 
for  $x, y \in adj(G)$  do
  if  $out\_link[x] = 0$  then
     $G_s[x, y] \leftarrow 1.0/|G|$ 
  else
     $G_s[x, y] \leftarrow G[x, y]/out\_link[x]$ 
  end if
end for
return  $G_s$ 
PAGERANK ( $G$ )
 $out\_links \leftarrow OutLinksForEachNode(G)$ 
 $G_s \leftarrow StochasticGraph(G, out\_links)$ 
 $G_t \leftarrow TransposeMatrix(G_s)$ 
 $result \leftarrow \emptyset * |G|$ 
 $p \leftarrow \emptyset$ 
for  $x, y \in adj(G_t)$  do
   $push(G_t[x, y]/|G|)$ 
end for
 $error \leftarrow 1.0$ 
while  $error > \varepsilon$  do
   $result \leftarrow \emptyset * |G|$ 
  for  $x, y \in adj(G_t)$  do
     $result[x] \leftarrow result[x] + G_t[x, y] * p[y]$ 
  end for
  for  $rank \in result$  do
     $rank \leftarrow rank * DF + \frac{1.0-DF}{|G|}$ 
  end for
   $error \leftarrow 0.0$ 
  for all  $rank, prev \in zip(results, p)$  do
     $error \leftarrow error + abs(rank - prev)$ 
  end for
   $p = result$ 
end while
return  $result$ 

```

- A heuristic considering the PageRank of each vertex.
- A heuristic using both PageRank and McSplitDAL.
- A heuristic using both PageRank and McSplitLL.

Please notice that both DAL and LL heuristics are computed dynamically, whereas the PageRank approach is applied only once at the beginning of the procedure.

3.3.8 McSplitX+PR

Within the framework introduced in Section 3.3.6, we exploit the ideas introduced by McSplitLL and McSplitDAL, enhanced by the integration of the PageRank heuristic. The union of these techniques produced two new versions of the McSplit algorithm, specifically referred to as McSplitLL+PR and McSplitDAL+PR.

Whilst the original McSplit idea was centered around the node degree heuristic, the subsequent variants were mainly based on McSplitRL, which used reinforcement learning as a vertex selection heuristic. However, whenever a tie is encountered, the heuristic falls back to the node degree for choosing a vertex.

Algorithm 16 The proposed McSplitX+PR algorithm optimizing a McSplitX implementation recalled in line 5

```

MCSPLITX+PR ( $G$ )
   $G_{ranks} \leftarrow PageRank(G)$ 
   $H_{ranks} \leftarrow PageRank(H)$ 
   $G_{sorted} \leftarrow SortGraph(G, G_{ranks})$ 
   $H_{sorted} \leftarrow SortGraph(H, H_{ranks})$ 
   $McSplitX(G_{sorted}, H_{sorted})$  return

```

We propose using PageRank as a standalone or tie-breaking heuristic, substituting it for the node degree. This approach is summarized by Algorithm 16. First, we apply the PageRank to classify the vertices of graphs G and H (in lines 1 and 2, respectively). Then, we sort the vertices following their ranks obtained by the previous classification (lines 3 and 4). Finally, we apply our McSplitLL or McSplitDAL (i.e., $McSplitX$, generically speaking) on the sorted vertices (line 5). This method leverages the Reinforcement Learning, to choose vertices dynamically along the search, and guarantees the use of the PageRank scores as a tie-breaker, particularly at the beginning of the algorithm, when the rewards are initialized to zero.

3.3.9 Experimental results

Experimental setup

We ran our tests on a workstation with an Intel Core i9-10900KF CPU and 64 GBytes of DDR4 RAM.

All our algorithms are written in C++, and we compiled it with GCC version 9.4. For McSplit and McSplitLL, we use the original versions obtained from the WEB and adapted for being used with our new heuristic. For McSplitDAL, we wrote an implementation that follows the ideas indicated by the authors [99] as we were unable to find an official version publicly available. In addition, since it has been proven to be beneficial, we borrow the graph swap idea from McSplitSD [103], and include it in all the variants of McSplit. Our core implementation adopts the C++ parallel version of McSplit. Unfortunately, not all versions may run in multi-threading mode. Thus, as we are interested in comparing our results with the ones gathered with the previous variants of McSplit, we present all results running all parallel versions with a single thread.

All algorithms were tested on a publicly available dataset [100]. We focused on the most extensive graphs, the ones with 100 nodes. Given the size of the set, we chose at least one experiments for each graph category, finally selecting 400 graph pairs.

Our tests are designed to evaluate the most practical aspect of all algorithms; thus, we evaluate their ability to find suitable solutions in a limited amount of time, instead of finding the optimal solution with an unlimited timeout. For each graph pair, we then record the size of the most significant solution found. We compare the different methodologies in terms of their capacity to find the largest solution in the slotted time.

We fixed the timeout to 60 seconds for each experiment. This timeout has been selected because experimentally McSplit often finds an effective solution along the first recursion path and it improves it only sporadically. Figure 3.11 plots the typical growth of the solution size with respect to the number of recursions. We can see that at the beginning (within a few thousand of recursions, usually performed in less than one second in our setup) the solution size increases very rapidly. Unfortunately, after the first few seconds, the solution grows slowly as most of the time is spent

searching the enormous solution space. In orange, we highlighted the solution size at the end of the recursion process. Please, notice that the number of recursions is reported on the x-axis on a logarithmic scale.

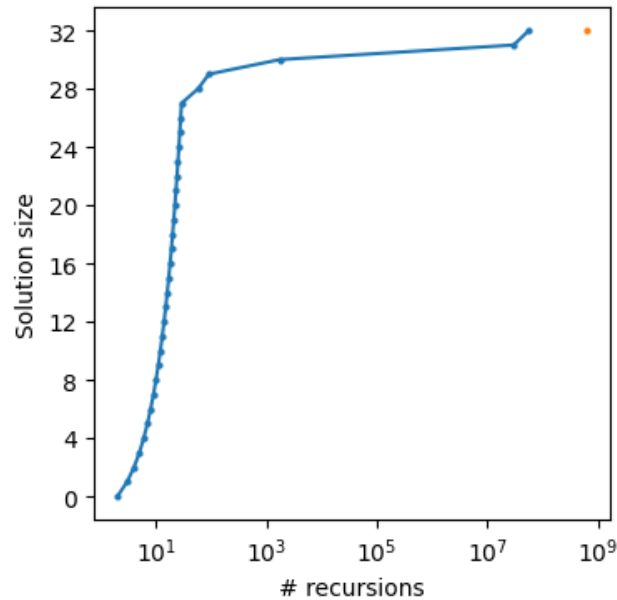


Fig. 3.11 Typical behavior of the effectiveness of the original implementation of McSplit. The size of the solution often increases rapidly in the first part of the process; then, the procedure is captured by local minima which slow down the convergence process and force the algorithm to visit enormous state spaces that do not improve the solution size. In orange, we can see the solution size at the end of the execution

Experimental evaluation

Figure 3.12 reports the number of graph pairs on which each method finds the largest MCS out of the 400 graph experiments run. When an MCS with the same size is returned by more than one heuristic (i.e., we have a *ex aequo*) that pair is assigned to all the methods returning that result.

It is straightforward to see that our PR heuristic, only applied to McSplit, McSplitLL, and McSplitDAL, easily outperforms the original strategies. Moreover, the fastest strategy, i.e., McSplitDAL+PR, finds the most significant solution in almost 300 cases out of 400.

Table 3.5, using no tie-breaker, shows the percentage of victories of all PR-improved strategies with respect to each original method.

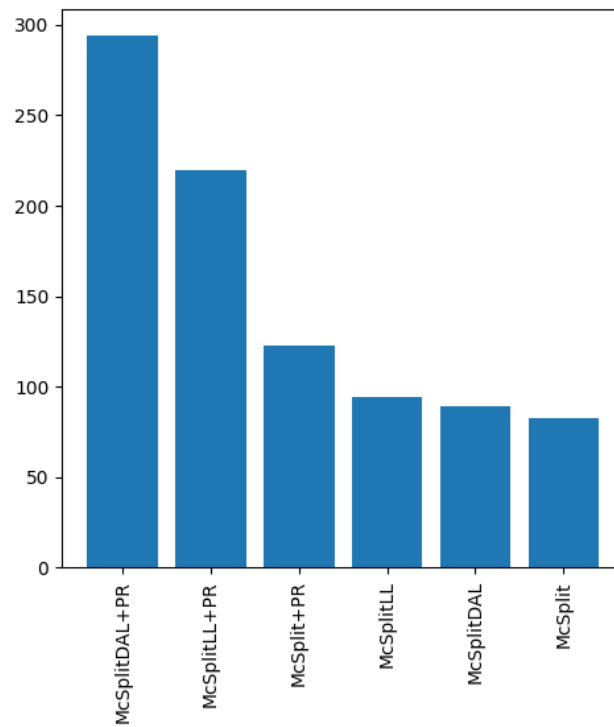


Fig. 3.12 The histogram plots the number of times each heuristic finds the MCS (i.e., the largest maximum common subgraph) on the 400 experiments. When a graph with the same size is returned by more than one method, each strategy is reported as a winner

Heuristics	McSplit+PR	McSplitLL+PR	McSplitDAL+PR
McSplit	64%	72%	77%
McSplitLL	60%	69%	76%
McSplitDAL	63%	72%	77%

Table 3.5 Percentage of instances improved by the PR methods (columns) over the original methods (rows), without breaking ties

Figure 3.12 and Table 3.5 focus on the number of experiments on which PageRank could return larger solutions than the original algorithms. Overall, they show that PR methods provide larger solutions for most of cases. However, we can also compare the size of the different solutions to understand the average improvements. To highlight the size of the results, we collected the size of the best solution found by each algorithm for every graph pair. To account for the natural variation in solution sizes between a wide range of instances of different complexity, we normalized all results with respect to the size of the subgraph found by the original McSplit algorithm.

In Figure 3.13, we show the average performance of our normalized heuristics. Due to the significant differences in solution sizes across instances, we plot a circular rolling average with a window size of 50 to better present the outcomes of our experiments. This strategy implies that each point on the plot represents the average normalized performance over a window of 50 consecutive tests. Due to the normalization, the original McSplit always returns solutions of size one, whereas all other methods almost always return more extensive solutions. Notably, PageRank demonstrates a distinct advantage over the degree heuristic. Moreover, McSplitDAL+PR and McSplitLL+PR methods consistently outperform their McSplitX counterparts in any batch of 50 instances and when they fall behind, they do not fall behind by a large amount.

The heat-map in Figure 3.14 shows the relative performance across all combinations of the algorithms. For each method on the vertical axis, the results are individually normalized with respect to the results of the algorithm on the horizontal axis; then, all the normalized values are averaged together.

From the map, we learn that McSplitDAL+PR exhibits an average improvement of 6% over McSplitDAL, McSplitLL+PR yields solutions that are 4% larger compared to McSplitLL, and McSplit+PR produces solutions 3% larger than McSplit. These results suggest that PageRank is an effective standalone heuristic, providing even more significant benefits when used as a tie-breaker on top of more complex Reinforcement Learning rewards.

It has to be noticed that in our testing, the McSplitDAL policy is not always better than the McSplitLL, unlike what was observed by Liu et al. [99]. This result is likely due to our different evaluation methodologies. However, McSplitDAL+PR

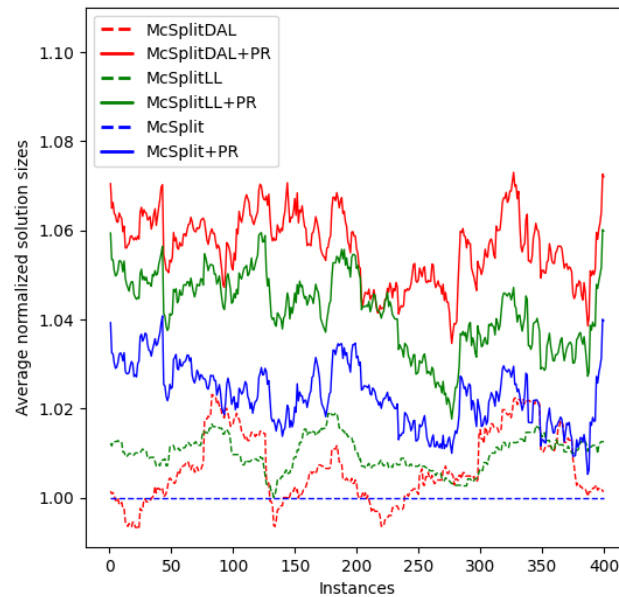


Fig. 3.13 A circular rolling average (with a window width of 50 consecutive tests) of the sizes of the solutions obtained by the McSplitX and McSplitX+PR algorithms on each instance. All values are normalized with respect to the results obtained by the original McSplit

benefits from the PageRank heuristic, convincingly outperforming both McSplitLL and McSplitLL+PR by 6% and 2%, respectively.

In Figure 3.15 we present a comprehensive comparison of the solution sizes achieved by each McSplitX+PR method and its corresponding McSplitX counterpart. For each instance, a dot is reported to show the size of the solutions found by the two algorithms. By removing the need for the rolling average, this scatter plot offers a better view of the results of the individual instances. Notably, the PageRank heuristic is the winner in most cases, particularly in the McSplitDAL+PR variant. Upon careful examination, it becomes evident that the average performance of the McSplitX methods is influenced by a few outlier instances that exhibit exceptional results. However, in contrast, McSplitX+PR consistently demonstrates improved performance across the entire range of instances.

3.3.10 Conclusions and future works

In this section, we saw an improvement over the resolution of the Maximum Common Induced Subgraph problem. Starting from a state-of-the-art algorithm called McSplit,

McSplit	1.00	0.99	0.99	0.98	0.97	0.95
McSplitLL	1.01	1.00	1.00	0.99	0.98	0.96
McSplitDAL	1.01	1.00	1.00	0.99	0.97	0.96
McSplit+PR	1.03	1.02	1.02	1.00	0.98	0.97
McSplitLL+PR	1.05	1.04	1.04	1.02	1.00	0.99
McSplitDAL+PR	1.07	1.06	1.06	1.03	1.02	1.00
	McSplit	McSplitLL	McSplitDAL	McSplit+PR	McSplitLL+PR	McSplitDAL+PR

Fig. 3.14 The relative performance of the McSplitX and McSplitX+PR methods. For each row, we report the average improvement relative to the respective column. Darker blue colors highlight the size improvements

and its recent variants (namely McSplitLL, McSplitRL, and McSplitDAL). we propose a family of Branch-and-Bound algorithms called McSplitX+PR.

The original McSplit algorithm uses a node degree heuristic to select the vertices of the graphs during the recursive search. McSplitRL and its derivatives use rewards obtained through Reinforcement Learning, but still enforce the node degree to break ties. We propose the McSplitX+PR algorithm family, namely McSplit+PR, McSplitLL+PR, and McSplitDAL+PR, to replace the original node degree heuristic with the ranking produced by the PageRank algorithm. PageRank, famously known as the former algorithm behind the Google search engine, generates more effective node orderings compared to the degree of vertices, as it prioritizes nodes that are easier to reach across multiple hops rather than just in the local neighborhood, effectively differentiating them over more categories than the original heuristic.

Using publicly available graph pairs, we conducted experiments on both the McSplitX+PR and McSplitX families. We mainly focus on finding the best solution within a limited time to simulate real-world scenarios. Our results indicate that all McSplitX+PR algorithms consistently outperform their McSplitX counterparts, with McSplitDAL+PR yielding the most effective solutions than the other strategies.

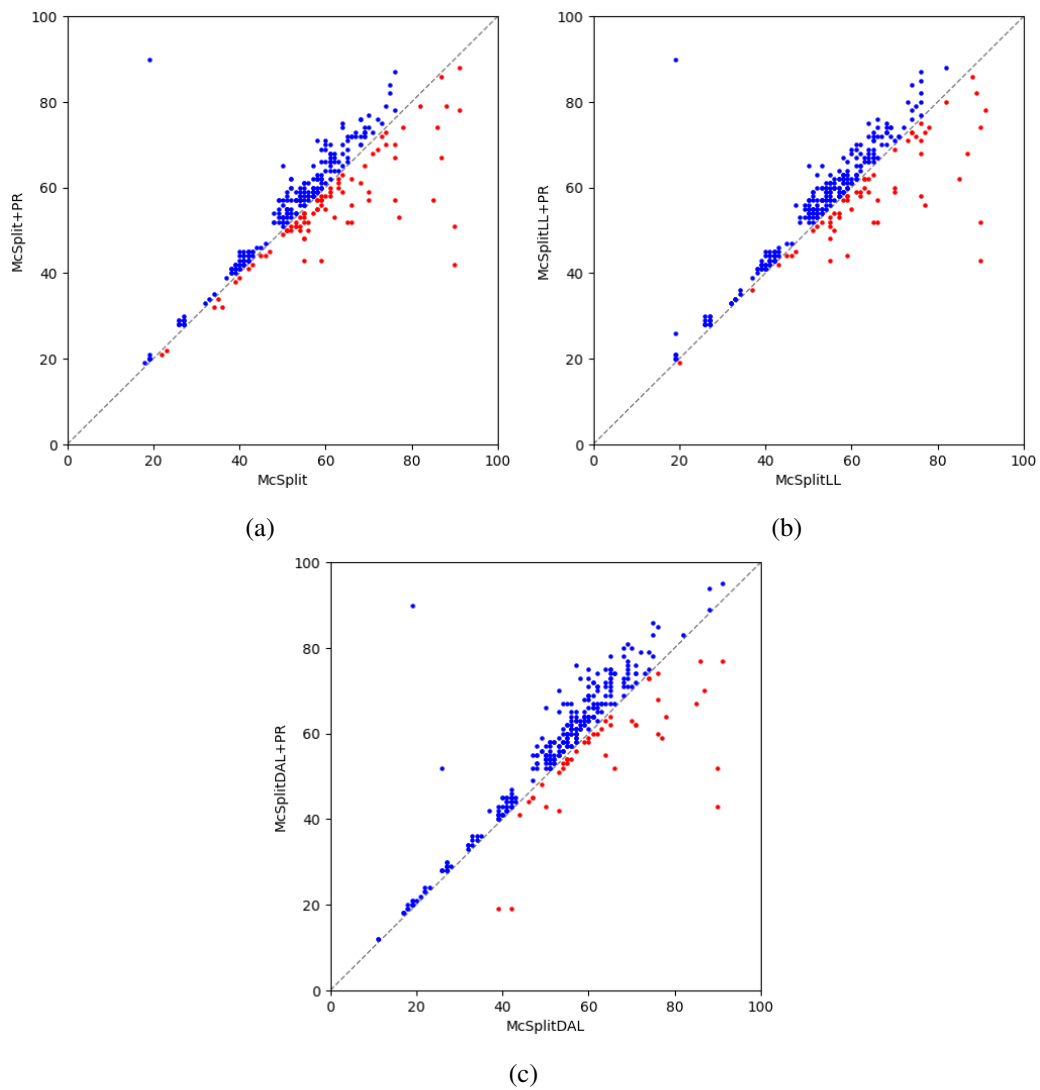


Fig. 3.15 The dispersion of the points above the main diagonal shows that McSplitX+PR finds more extensive solutions in the vast majority of the cases

Among the possible future works, we would like to mention the necessity of studying the multi-threaded versions of the above tools. In this work, this analysis has been limited by the fact that not all the considered tools were initially implemented with multi-threading capabilities. Consequently, one of our targets is to improve the above heuristics obtaining uniform scalability on multi-core architectures.

3.4 Conclusions

This chapter has come to an end. In it, we discussed about the porting of a well-known graph coloring algorithm, compared with the different state-of-the-art approaches, such as GM and its porting on GPU through the ATOS approach.

Moreover, we saw improvements on McSplit using the PageRank algorithm to classify nodes. A work has been sent to be published comparing McSplit with other node classification algorithms. However, the work of that paper does not appear in this thesis, as it is yet to be accepted.

Now, it is for the last work I will talk about in this thesis: an AI approach to classify objects inspired, once again, by the work of the human brain and a bit of psychology.

Chapter 4

A Core knowledge-based AI

In this chapter, I will talk about the work accepted in [110]. This work has been developed in collaboration with Alberto Tonda from INRAE (Institut national de la recherche agronomique). Here, I will present a new approach in Artificial Intelligence development, inspired by the concept of Core knowledge, a philosophical and psychological theory that suggests that we are born with some concepts already developed in mind, working as building blocks for our future knowledge.

Artificial Intelligence (AI) is an umbrella term that covers several different techniques, ranging from rule-based systems to Machine Learning (ML), from Reinforcement Learning (RL) to Evolutionary Algorithms (EAs). In the last decade, AI algorithms have set important milestones in a wide range of domains, such as natural language generation [111], image classification [112], prediction of protein folding [113], and even videogame puzzles [114]. Interestingly, almost all such successes have been attained by ML algorithms and often by the most recent generation of artificial neural networks, known as Deep Learning (DL).

DL approaches can learn predictive models directly from large data samples. Still, they suffer from essential limitations [115], such as *brittleness*, the unpredictable and undesirable behavior for given samples, *opacity*, the impossibility of explaining the overall predictions of the model, and *limited generalization ability*, the poor prediction quality for out-of-distribution samples.

Most of these issues stem from the black-box nature of the models, which cannot be easily inspected and whose behavior cannot be verified by human experts. A few sub-domains of AI are currently exploring different possible solutions. The

eXplainable AI (XAI) community [116] is developing techniques to make black-box models more human-readable, for example using concept bottlenecks [117] or visualization techniques able to highlight the features that ML/DL models are using to take decisions [118, 119]. On the contrary, neural-symbolic (NeSy) approaches [120] aim to combine modern neural-network models with classic symbolic AI, capitalizing on both advantages.

A different research direction, proposed in [121], is to build AI systems exploiting principles similar to human *core knowledge*, that is, a small set of innate capacities identified by cognitive psychologists [122]. Examples of core knowledge include evaluation of quantities, identification of agents, and prediction of movement. Instead of starting from a *blank slate* informed only by the available training data, as in the case of current state-of-the-art ML, AI algorithms may start with a limited amount of specialized hard-coded capacities, compose and combine them to solve tasks and use only a limited amount of training samples. In principle, algorithms that learn in a way that is more similar to humans could require less training data to complete tasks while providing a white-box, interpretable explanation of their behavior.

In this paper, we propose a first step towards an evolutionary AI approach inspired by core knowledge, where the EA is used to create a high-level, concise description of a simple 2D video recorded from a video game. Using a small number of primitive concepts, such as patches tracked over frames and simplified laws of motion to predict their expected behavior, the proposed approach can generate a compact, correct video description. Our approach first groups patches that behave coherently into objects. Objects are then grouped into classes, i.e., groups of objects that share similar behaviors. The interactions between objects belonging to specific classes are then described through rules. In the proof of concept presented in this work, the identification of objects is performed through simple heuristics, whereas the association of objects to classes and the creation of rules is delegated to the EA engine.

The proposed approach is evaluated on two popular benchmarks, Pong and Arkanoid, and it has been proven to be able to explain the interactions appearing in short video-game videos, generating accurate and human-readable descriptions from a relatively small amount of data. Starting from these initial, promising experimental results, future works will focus on exploiting the rules obtained by the proposed

approach to actually play video games, providing a white-box explanation for each action the AI agent takes.

The rest of the paper is organized as follows. Section 4.1 summarizes the necessary concepts to introduce the scope of the work. Section 4.2 describes the framework of the proposed methodology. Our experimental analysis is detailed in Section 4.3. Finally, Section 4.4 summarizes our findings and outlines possible future works.

4.1 Background

4.1.1 Core knowledge

Core knowledge is a psychological theory of human cognition [122] postulating a small set of innate cognitive capacities all humans are born with. The source of this core knowledge is uncertain, but it is currently believed to be the process of natural selection. As other animals can perform complex tasks or movements since their early infancy, humans possess similar capacities that can be exploited, for example, to learn a language from a relatively small amount of sample sentences, as the ones typically experienced during childhood [123]. All animals, however, have the potential to learn new skills or concepts through experience, and distinguishing core knowledge from learned skills is not a straightforward process. Cognitive psychologists tackled this challenge by either evaluating the performance of human groups that are culturally isolated or by focusing on toddlers and infants. Notable examples include the study of arithmetical intuition among indigenous populations of the Amazonian rainforest [124, 125], and the assessment of quantity in 6-month-old infants [126]. While providing a complete overview of the different perspectives in psychology is outside of the scope of this paper, it is worth noting that there are alternative theories to core knowledge, for example, connectionism, and that the long-standing question of *how humans learn* has been tackled through a vast number of viewpoints, see [127] for a discussion.

Innate capacities that are part of core knowledge have been tentatively clustered into different systems [128], depending on their nature. Each system focuses on a set of principles used to individuate the entities in its domain and to support inferences about the entities' behavior, such as: (i) inanimate objects and their mechanical inter-

actions, (ii) sets and their numerical ordering, addition, and subtraction relationships, (iii) places in the spatial layout and their geometric relationships, (iv) agents and their goal-directed actions, (v) potential social partners and social group members.

Besides the detailed identification of each innate capacity, studying core knowledge can open further insights into human cognition. For example, the human ability to extrapolate knowledge to unknown but related domains could stem from core knowledge systems combined through the principle of compositionality [129].

Unsurprisingly, this line of research caught the attention of a part of the AI community. In [121], Chollet outlines a novel research line for human-like AI, proposing a new dataset, the Abstraction and Reasoning Corpus (ARC). ARC comprises hundreds of tasks represented by static images, with only a few solved instances provided for each task, plus a test instance. Humans can infer the general principles required to solve the test instance from the examples provided. At the same time, the current state-of-the-art AI algorithms cannot, as they typically require more examples to learn. Following the theory of core knowledge, the skills required to complete ARC tasks are primarily related to geometrical and spatial intuitions without considering movement, agents, and their actions. Still, it can be argued that core knowledge evolved in a three-dimensional, dynamic world enriched with goal-directed entities. Consequently, attempting to complete tasks including agents could potentially be more beneficial. This gap motivates the present work, as we focus on inferring relationships between agents and objects in movement, designing an evolutionary AI framework specifically tailored to achieve this objective.

4.1.2 Related works

Mainstream AI is currently dominated by ML/DL approaches, exploiting massive quantities of data to train large, complex models for specific tasks. Nevertheless, in specialized literature, it is possible to find several AI systems that aim to be less data-hungry and more human-interpretable. The heterogeneity in terminology and domains makes it challenging to survey these approaches comprehensively. A good starting point is the overview given by Mitchell [130], where the author summarizes the state of the art in systems able to reason by analogy and traces a few possible research lines for the future.

Just as the approach proposed in this work aims to generate a set of rules, classic Learning Classifier Systems (LCS) [131, 132] can also create a set of rules, albeit for classification problems. The final classifier is human-readable, but most LCS systems exploit simple rules. A few research lines in the domain of LCS have put forward interesting ideas, including cognitive LCS able to build hierarchies of rules [133].

Fister et al. [134] focus on Numerical Association Rule Mining, where numerical attributes are handled without discretization. The authors present a historical overview and the main features of algorithms dealing with problems with categorical and numerical attributes.

Kumar et al. [135] concentrate on agents trained by meta-learning. These agents may sometimes acquire very different strategies from humans. The authors show that co-training these agents on predicting representations from natural language task descriptions and programs guides them toward more human-like inductive behaviors. Human-generated induction models add newly learned primitives containing abstract concepts that can compress description length. Co-training on these representations results in more human-like behavior.

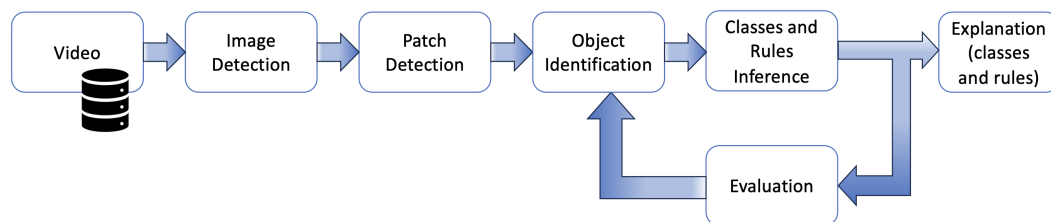


Fig. 4.1 Proposed pipeline for finding general rules describing interactions between classes.

Haase et al. [136] consider the projectile collision game Angry Birds on which the AI benchmark AIBirds is based. The authors investigate whether a qualitative approach to action planning under uncertainty proposed by Ge et al. [137] can be adapted to the domain of Angry Birds to identify targets hittable by multiple rebounds. The authors discover that the search space for solving the game is complexly structured, and a fine-grained decomposition is required, leading to high computational costs.

4.2 Proposed approach

We propose a novel EA-based methodology to create the description of a simple, two-dimensional video, identifying *objects* appearing in it, clustering them into classes, and generating rules describing their interactions. The process is organized as follows. First, we perform data analysis, gathering the positions of the segmented patches and their possible contact points in each frame. Then, we detect the interactions between patches, transforming them into objects and providing a physical status, with features such as speed and shapes identified by the patches forming that object. Then, an EA aggregates objects with a coherent behavior into *classes* and creates *rules* to describe the observed interactions between each other, ultimately building a human-readable description of the actions identified in the video. We deem EAs particularly suited to this kind of task, as they can efficiently explore vast search spaces in a relatively short amount of time, and the link between evolution and learning has been explicitly pointed out by Turing [138] among many others.

4.2.1 Objects identification

Figure 4.1 shows our approach to finding the classes and rules in our framework. Starting from a video featuring only 2D patches, different patches are detected in every frame through simple heuristics. The patches are then aggregated into objects. This step could be performed through several different approaches. We use an approach based on evolutionary optimization; in this first proof of concept, we opted for simple heuristics. The Gestalt principle of proximity [139] is thus employed to identify multiple instances of the same patch across subsequent frames.

After detecting the different patches, the algorithm assembles *objects* as sets of corresponding patches across frames. An object can be either static or dynamic. Static objects can be defined using simple rules mimicking humans' idea of persistence: (a) the object exists only in one position, and (b) the object may disappear, but if it reappears, it does so in the same position. Dynamic objects, on the other hand, are clusters of patches that move through the screen. There are usually few moving elements in the analyzed video streams, as we discuss more in detail in Section 4.3.

We model concepts following basic laws of physics, allowing the approach to detect events inspired by physical interactions. For instance, in Arkanoid, the contact

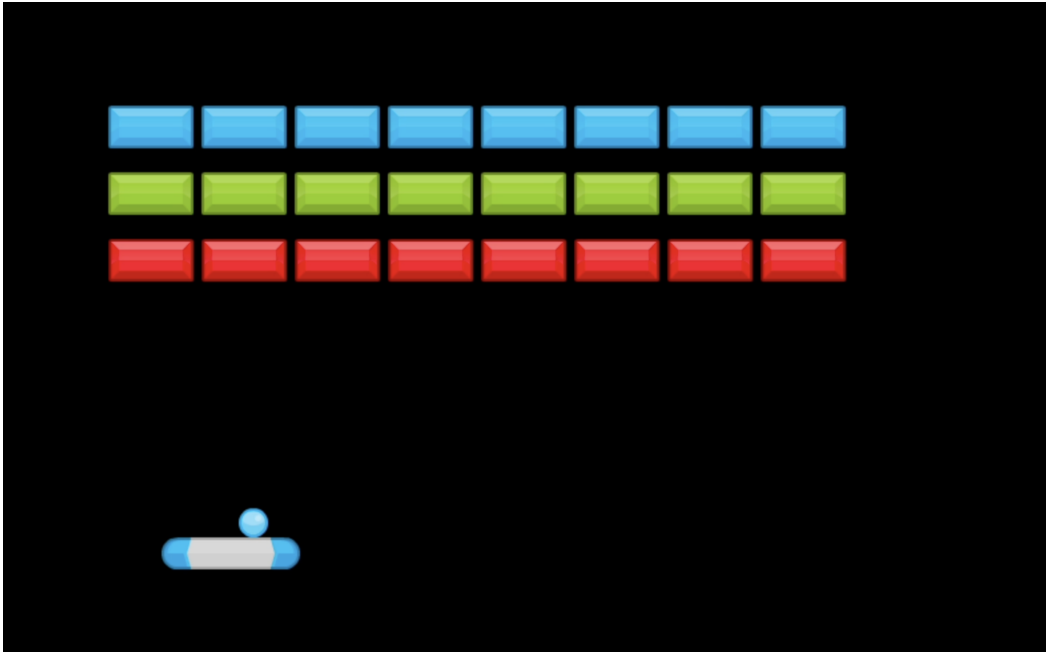


Fig. 4.2 A video frame of Arkanoid shows the instant in which the ball hits the user's paddle and is about to bounce back.

between two patches may be detected as an interaction, providing a cause-effect relationship between events. More specifically, some patches are recognized as the object "ball" and others as the "paddle". Then, when the ball collides with the paddle (Figure 4.2), the velocity on the y -axis of the ball changes sign, and the velocity on the x -axis changes based on the angle formed by the ball and the center of the paddle. Such fundamental interactions are the core knowledge of the system, i.e., the idea of "collision", the dynamic of a "bounce", and the appearance and disappearance of an object. Moreover, it could be noted that even a cause-effect relationship is part of the core knowledge, mimicking how humans explain the world.

More generalization becomes possible as more physical events are discovered in the video frames, as the same rules may explain several other events. Generalization can also reinforce some of the previous hypotheses and rule out others. For example, on the one hand, a video not including a bounce with the top "wall" may lead to an indeterminism in the classification of the wall itself. To the analysis, the wall may as well not exist or be a background object. On the other hand, if there is an interaction between the ball and the wall, the effect is discovered, and the wall is correctly classified.

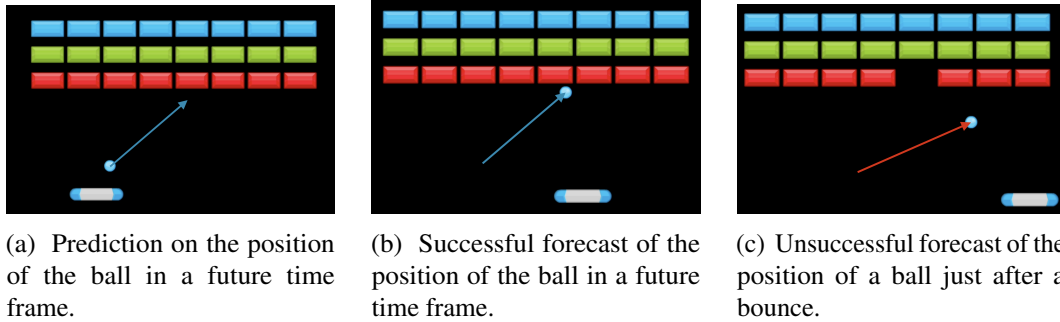


Fig. 4.3 A visual analysis of the correction of the ball velocity performed by our framework.

Patch detection, however, presents a few technical obstacles. In particular, the most challenging issue is an aliasing problem. Our approach involves knowing the velocity of patches at each frame. However, all our data is related to pixels, which can be either *belonging* or not belonging to the patch in a binary way. Thus, analyzing differences frame-by-frame may not yield a satisfactory result, as this procedure may detect spurious accelerations due to the discretization. To overcome this difficulty, we provide knowledge based on the first law of physics, i.e., an object not subject to any forces has a constant velocity. Thus, if we assume a uniform speed between two frames, the precision of detecting the actual rate of the object increases, as does the distance between frames. Similarly, a larger distance between positions, obtained by expanding the time difference between frames, can provide a more precise average velocity. However, we shall ensure that the hypothesis is verified; thus, each time we expand the time difference, we check if the ball position on the frames in between is well-approximated.

Figure 4.3 shows an example of the proposed prediction strategy. First (Figure 4.3a), we know that the ball has a certain speed and trajectory. Then, we move to the next frame until the following condition is true:

$$d(p, e) > \varepsilon \quad (4.1)$$

where d is the Euclidean distance, p is the actual position of the ball, and e is its expected position, meaning that the distance between the actual position of the ball and its predicted position diverges significantly. Ultimately, we update the speed with the best value we have. ε is a user-defined value, typically small, i.e., but requires more precision than one pixel, which can be, for instance, 4.1 pixels. After the speed update, we can remove spurious accelerations from the events for each

patch. Reducing the number of events increases the visual analysis's accuracy and improves the computation time during the learning phase. Finally, we formalize the events involving all patches and confirm the corrections and event filtering.

4.2.2 Evolutionary learner

The last step of our process is a learning phase, performed by an EA, that aggregates the objects detected by the heuristics into classes (i.e., groups of objects that behave similarly), and then describes the interactions between objects belonging to classes using rules.

Candidate solution

We detect two distinct elements: *Classes* and *rules*. A class is a coherent aggregate of objects. A rule is a cause-effect relationship, where the cause is an interaction between two objects, and the effect is an alteration of the state of some target objects, which might or might not have been directly involved in the interaction. Classes and rules are ontologically dependent on each other, i.e., a class's existence depends on the presence, or not presence, of the interactions with other objects. For instance, the Arkanoid game's top, left, and right walls trigger a bounce in the ball; however, their state does not change in any way. Finally, some events may not be explicable from what is observed. An example is the horizontal movement of the paddle in Arkanoid, which depends on the user input and that no observer can explain with a rule.

However, according to the video analysis, the EA finds a set of classes and rules derived from the "perceived" *events*, minimizing the number of rules and entities involved in the explanation.

Each candidate solution can have an arbitrarily large number of classes and rules, with a minimum of one class and one rule (describing the interaction between objects belonging to the same class). The fitness function, presented in the following paragraph, applies a selection pressure for simplicity to push for the minimal number of classes and rules needed.

Fitness function

Intuitively, the evaluation of an explanation includes terms for: (i) its accuracy for the behavior of the classes identified in the video; (ii) its complexity, with a preference given to the simpler description between two approximately equally accurate solutions, with an idea similar to Occam's razor. Thus, the fitness function defined for this optimization problem is:

$$F(I) = \neg E(I) + \alpha \cdot (C(I) + R(I)) \quad (4.2)$$

where I is a candidate solution, $\neg E(I)$ is the number of unexplained events, $C(I)$ and $R(I)$ are the total number of classes and rules in the solution, respectively, and α is a user-defined weight regulating the relative importance of the second part of the equation. For example, setting α to a small value means that a solution with a larger amount of classes and rules able to explain a more significant portion of the events will be considered more promising than a solution with fewer rules but unable to explain as many events. The value of $F(I)$ must be minimized.

Genetic operators

Given the structure of a candidate solution, we define genetic operators that are applied to its different parts. Each operator may act to modify either the set of classes or the set of rules in each explanation. In particular, for the set of classes, the following operations can be performed:

- **Class mutation:** A randomly selected object is moved between two randomly selected classes in the same explanation; if a class ends up with no objects, the class is removed from the explanation.
- **Class removal:** A randomly selected class is removed, and its objects are randomly distributed among other existing classes with uniform probability; all the rules in which the class is involved are discarded.
- **Class addition:** A new class is added to the explanation, and a randomly selected object is moved to the newly created class.

Regarding the rules, the following specialized operators can be applied:

- Rule mutation: An existing rule is mutated; the mutation may involve the target class, the interacting classes, the cause, and the effect; in all cases, a new randomly selected element is selected to replace the current one.
- Rule removal: A randomly selected rule is removed.
- Rule addition: A new randomly generated rule is created.

4.3 Experimental evaluation

We tested the proposed approach on Pong and Arkanoid (also known as Breakout), two extremely popular two-dimensional video games [140]. These experiments aim to test our strategy on puzzles based on similar concepts but different layouts.

We consider Pong the easiest game to model, as it involves a ball bouncing on both the walls and the two paddles on each side; each object can send the ball back. Paddles are user-controlled and can be moved vertically up and down. Each player's target is to hit the opposite wall with the ball, getting beyond the opponent's paddle and scoring a point. In single-player games, the computer can control one of the two paddles with a simple AI.

Arkanoid features a single horizontal paddle controlled by the player, a set of bricks, and a ball that can bounce on walls and bricks. Each time the ball bounces against a brick, the brick disappears, and the player's score increases. The game's goal is to make all bricks disappear by hitting them with the ball; missing the ball with the paddle and letting it hit the bottom of the screen causes a game over. The paddle is user-controlled and can be moved horizontally left and right.

We tested our approach with two different videos for each game. Each video is recorded with 60 frames per second. For Pong, the first video includes about 4,318 frames, and the second includes about 6,086. For Arkanoid, the videos are 2,335 and 12,082 frames long. Moreover, each experiment was repeated 30 times with 30 different initial seeds for the random generator to check the convergence ability of our technique. Each video is part of a *training test case*. Ideally, with sufficient events in the video, we show that converging to a shared knowledge of the game is possible. All experiments are run on a server using an Intel(R) Xeon(R) Gold 6238R CPU @ 2.20GHz, equipped with 256 GB of RAM. All the code and the data

necessary to reproduce the experiments are freely available on a GitHub repository at <https://github.com/to-be-disclosed-after-revision>.

4.3.1 Implementation

The scene analysis focuses on identifying separate patches inside and across video frames. Many image-segmentation techniques have been proposed to detect elements in images, and the best choice is often application-specific. In the current work, we select OpenCV [141], which provides a simple approach to analyzing a simple video showing a typical game played by a human player. Within OpenCV, we detect patches through image similarity: If the RGB channels of an image are above a threshold, then the image is detected within the current frame. The threshold may vary due to the image's shape and is mainly found through visual inspection with approximating supervised tests. We perform this operation for every image used. Notably, the detection of patches is the paradigmatic activity that could be delegated to a state-of-the-art neural network.

The EA used in the experiments employs a classic $(\mu + \lambda)$ replacement scheme and a tournament selection for choosing the individuals for reproduction. The evolutionary library used in the experiments is `inspyred` [142]. For all the following experiments, we use a population size of $\mu = 1,000$ and an offspring size of $\lambda = 1,000$ individuals. The tournament selection employs $\tau = 2$, and the termination condition is set on 2,000 maximum generations, with an early stop if the best fitness does not improve for 100 generations. When a new candidate solution is to be produced starting from a parent solution, a single genetic operator is selected with uniform probability among those previously described. We used the value 0.001 as α in Equation 4.2; this way, α is used to make the number of classes and rules a secondary objective. For the value of ε in Equation 4.1 we set the value to 2.1, which worked for both Arkanoid and Pong.

4.3.2 Case study 1: Pong

In the game of Pong, a ball bounces against two paddles that can move vertically, as shown in Figure 4.4.

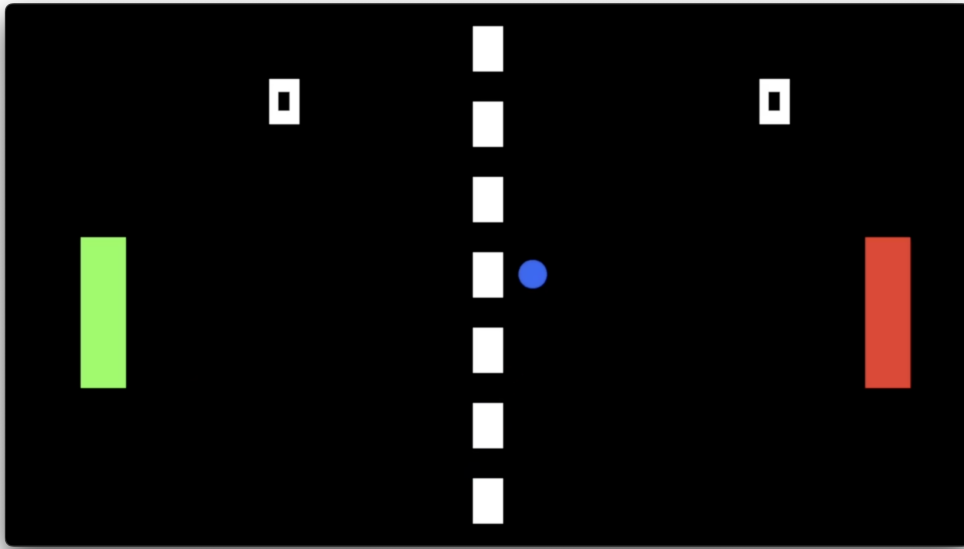


Fig. 4.4 An image of the Pong game. The net (white stripes) divides the playing court in half; the two user-controlled paddles play against each other. The blue ball is represented in the middle of the court, and the scores are reported on the top of the screen.

The ball starts with an initial velocity on the x-axis $v_x \neq 0$, and the paddles should catch the ball. Once the ball touches a paddle, it bounces. There are two variants of this game. In the first one, ball bounces are perfectly elastic; in the second one, the new angle of the ball depends on the distance from the center of the paddle. We experimented with both variants, as they deliver slightly different results.

In Figure 4.4, we show an example of the Pong game played by two players: the green and the red paddle. In the video analysis, when the ball touches one of the white stripes, it disappears (meaning that the video analysis has no concept of the permanence of an object). Consequently, the algorithm finds the following rule: The ball bounces when the ball touches one of the white stripes (grouped in a class). This effect is seen in Figure 4.5, showing that the algorithm adds the rule for the ball's disappearance. In particular, starting from different complexity values, the algorithm can improve over the solution until it finds the final complexity of 227.006 within 20 generations at most.

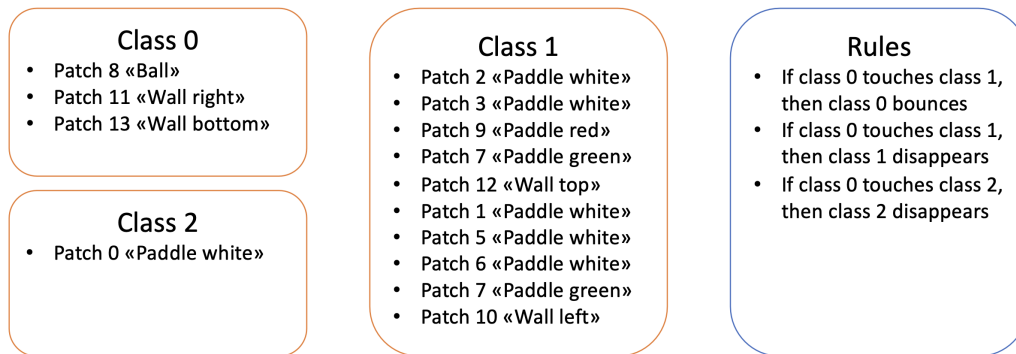


Fig. 4.5 The Pong puzzle: The classes and rules discovered by our strategy.

4.3.3 Case study 2: Arkanoid

Arkanoid is a game in which a player controls a paddle at the bottom of the screen. The paddle moves only horizontally (left and right) to manage the bounces of a ball. The other elements in the game are bricks. A brick disappears when the ball hits it. The goal of the game is to make all bricks disappear. In our case study, we have three lines of bricks, and bricks of different colors represent each line. Figure 4.6 shows a video frame representing an intermediate development of a game.

The ball bounces elastically against the walls and the bricks. Colliding against the paddle generates a bounce that follows the same rules we analyzed with Pong. Arkanoid is more challenging to interpret for our learning program. This is due to the higher number of patches in the frames and the more significant number of rules required to interpret all events. In particular, our instance of Arkanoid can be described using 26 patches, whereas Pong requires only 3 (without the net in the middle) or 10 (with the net) patches.

As shown in Figure 4.7, the learner can separate the ball from the other patches by finding the classes “ball”, “blocks”, and “misc”. However, most of the blocks belonging to the miscellaneous class are artifacts. Indeed, they disappear when the ball touches them, even if it never bounces back. Moreover, as the bottom wall is never touched in this game, the learning algorithm has no experience with its interactions and places it in a random class.

This behavior is similar to the one we discovered analyzing Pong. In both games, we converge toward a very good set of rules, perfectly describing the basic principles

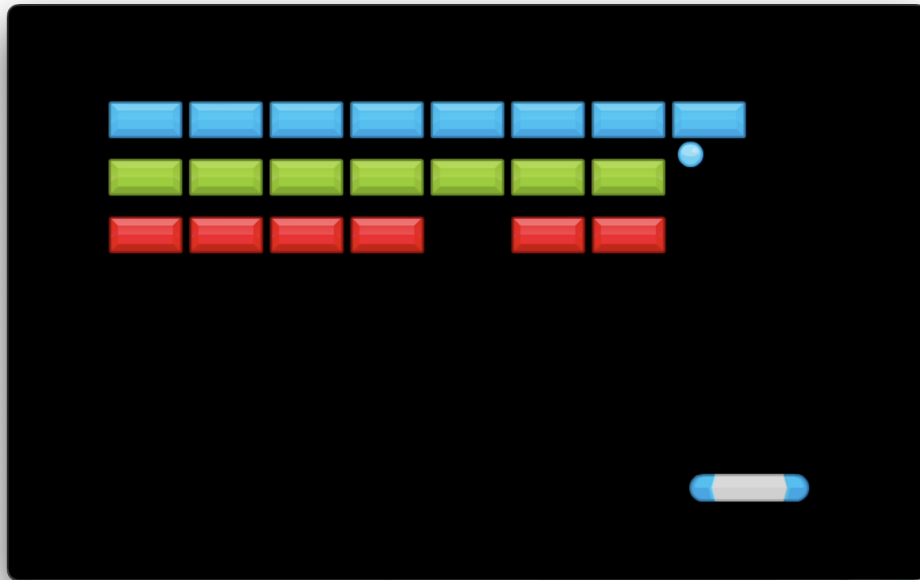


Fig. 4.6 A video frame of an ongoing Arkanoid game, where some bricks have already been hit by the ball and disappeared from the image.

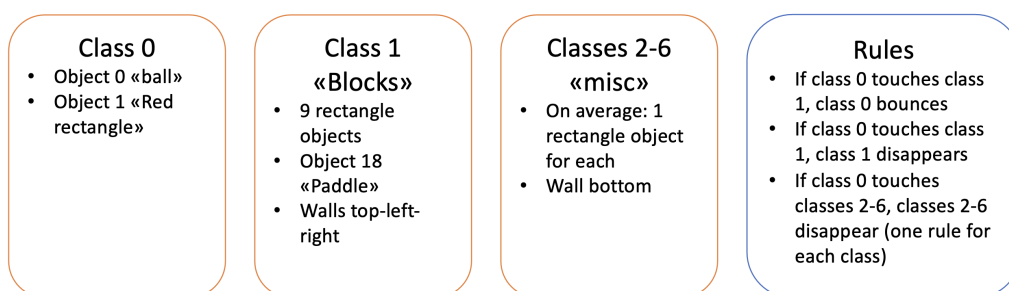


Fig. 4.7 The Arkanoid puzzle: The classes and rules discovered by our strategy.

of the games. For Arkanoid, such rules can be described as “the ball bounces”, “all blocks share the same behavior”, and “each block disappears when hit by the ball”.

4.3.4 Behavioural considerations

We analyze two games with several similarities. They both involve balls, paddles, and bounces, but in Arkanoid, there are bricks, and those objects may disappear during the game’s evolution.

Our solutions converge as expected on both game videos, leading to a unique and reasonable explanation. Table 4.1 shows that the algorithm converges to the same value even using very different starting seeds. In the table, seeds are selected randomly. The first column shows the seed we used. Then, for each game, the first column reports the fitness of the best individual after the first generation, and the second one presents the best individual in the last generation. However, the number of generations needed to converge varies significantly. For instance, for the Pong game our approach may require more than 40 generations to converge to the solution with fitness shown in the fifth column of Table 4.1. Arkanoid is the most challenging game to analyze. On average, converging requires more than 70 generations, with minor differences in the convergence rate depending on the initial random seed used to find the solution. By exploring the solution space, we noticed a typical pattern: Our approach tries to minimize the number of classes despite the number of rules involved. When an object is moved between classes, the number of rules increases, creating false explanations. However, as one of our targets is to minimize the number of rules, the ones that do not provide any proof are automatically discarded after three or four generations from the generation in which they are introduced.

In addition, we noticed that the video analysis process is extremely important to reaching a good explanation. As there are different formats to represent a video stream, we used the H265 codec [143] wrapped in MP4 files for compatibility. Consequently, some differences may arise by analyzing the video using different codecs, which may create video artifacts. Moreover, we found that OpenCV behaves differently when used with Python or C++. Currently, all our experiments adopt the Python version of OpenCV; its parameters for detecting images are fine-tuned for this version. However, in our last set of experiments, we found that the C++ counterpart of OpenCV is much more accurate and may lead to better results. We consider these

Seed	Pong Evolution [Generations]		Arkanoid Evolution [Generations]	
	First	Last	First	Last
123456	237.006	227.006	171.020	150.014
42	238.007	227.006	167.005	150.014
256	236.007	227.006	157.007	150.014
190283	236.007	227.006	160.005	150.014
328	239.005	227.006	159.005	150.014
715321	234.005	227.006	161.007	150.014
1	235.007	227.006	160.007	150.014
0	228.005	227.006	171.005	150.014
2	234.006	227.006	161.007	150.014
10000	235.005	227.006	161.004	150.014
444	235.007	227.006	161.007	150.014
711	234.005	227.006	165.007	150.014
8125002	235.007	227.006	165.004	150.014
30	234.008	227.006	164.004	150.014
59	235.007	227.006	164.006	150.014
100	239.006	227.006	157.007	150.014
99	233.006	227.006	166.004	150.014
999	234.004	227.006	160.004	150.014
999999	229.004	227.006	159.008	150.014
13	235.008	227.006	167.006	150.014
75	237.007	227.006	159.007	150.014
61	233.008	227.006	167.004	150.014
71	235.006	227.006	163.008	150.014
915	235.004	227.006	164.004	150.014
627	233.006	227.006	157.006	150.014
498	242.005	227.006	166.006	150.014
186	236.004	227.006	168.006	150.014
216	234.006	227.006	161.006	150.014
311	235.008	227.006	172.006	150.014
618	234.008	227.006	164.008	150.014

Table 4.1 The evolution of Pong and Arkanoid games respectively: for each seed, we show the fitness of the best individual in the first generation and the fitness of the best individual in the last generation

experiments too preliminary to be conclusive and reported in the current paper, but believe they can be a part of our future work and implementation effort.

4.4 Conclusions and future works

We present an approach to explain a 2D video as a set of objects, classes, and rules. We apply it to two simple but widespread games, Pong and Arkanoid. The approach provides a human and machine-readable description of the videos and a resulting explanation for their models. We show that, even with a limited training set, the process can differentiate classes and abstract rules involving them. Thus, the approach builds a believable explanation of the rules followed by the various classes.

This work is the first step in a research line to build AI systems inspired by core knowledge and exploiting evolutionary computation to aggregate the basic information provided by hard-coded algorithms. The following step in developing such a system may move in different directions. The first possibility is to leverage the descriptions created by the approach to evolve optimal video-game plays, offering a potentially more robust alternative to DL-based solutions. Another alternative is experimenting with combinations of EAs with other search algorithms, such as Novelty Search [144]. Finally, we mention the possibility of finding more free-form structures to aggregate the outputs of the algorithms used as part of the core knowledge provided to the approach.

4.4.1 Towards the future works

From a technical practical, we could significantly improve the algorithm's speed by using a compiled programming language such as C, C++, or Rust. This has been done after the acceptance of this paper, and partially left as future work for future thesis students. From the current partial results, as mentioned in Section 4.3.4, using the C++ version of OpenCV shows significantly improved results by increasing the accuracy of the video analysis.

Chapter 5

Conclusions

This is the final chapter of the thesis, and here lies the conclusion of my Ph.D. journey. Before writing my final words on this thesis, however, I would like to remark the technical achievements reached by me, my supervisors and my colleagues.

5.1 A short wrap-up

In Chapter 2 I presented three works aimed to improve the hardware testing state-of-the-art. First, I presented a work on VCD files [18], with the focus on improving the feasibility of custom analyses of testing programs applied on SoCs, exploiting multithreading to improve significantly the computation time over three different types of analysis. Later in the same chapter, I presented a work on the toolchain [20] we developed together a series of tools that work together to provide a more in-depth analysis of the metrics analyzed by the VCD analyzer. In particular, we developed a tool to filter out some of the results provided by the VCD file; later, we focused on analyzing the uniqueness of the stress provided by each testing program, with the set tool. The set tool can also merge subsequent results of the VCD tool, providing the effective coverage also from bits that could be stuck in a single test program. Moreover, we developed a tool to weight the results from the analyses, effectively ranking the various faults based on their location on the SoC. Another tool we developed classifies the faults based on the module they are in, providing an effective coverage of the whole module on the SoC. Finally, we developed a program to visualize on an image the fault coverage provided by the testing programs. The third

work presents a new metric, called connectivity [19], to help increasing the fault coverage of a test program trying to avoid expensive fault simulations, providing a not exact, but easy to compute, approach that helps test engineers to develop their software-based tests correctly.

Then, in Chapter 3 we improved two existing algorithms on graph computations. The first one is a GPU implementation of the JPL algorithm [65], written in C++ and CUDA. This uses a semi-random node selection to color the nodes in a graph, effectively coloring nodes that are surely not close to other nodes with the same color. Moreover, in the second section I present our work on McSplit and the use of PageRank to improve its results, together with its state-of-the-art variants [66].

Eventually, in chapter 4, I presented a new approach to the AI task [110], in which we developed an agent able, with some small mistakes, to explain the actors in two video-games and the rules to which they are subject. This work is still in progress and is going to be presented in GECCO 2024.

5.2 Some final words

First of all, I would like to thank every person that worked with me and supported me. I will start with my supervisors, Stefano Quer and Giovanni Squillero, that saw me at my worst with some works that I chose not to write here... and probably with this thesis itself.

I am a firm believer in collaboration, and I had some of the best colleagues I could have hoped for. I will start with the ones that are present on this thesis: Francesco Angione, Lorenzo Cardone, Gabriele Filipponi, Giusy Iaria. Together with them, I would also like to thank professor Paolo Bernardi, who also looked after me more than he had to, and Alberto Tonda from INRAE. I would like to thank my thesis students, starting with the known ones: Alessandro Borione, Salvatore Licata and Marco Porro. However, we shall never forget the ones that were not mentioned before: Enrico Carraro and Thomas Madeo (after one year, will our paper be accepted?). I would also like to thank my colleagues at Lab 3 that I did not mention: Mohammadreza Amel Solouki, Tommaso Foscale, Giorgio Insinga, Nima Kolahi Mahmoudi, and Annachiara Ruospo.

Of course, my parents are also in the list: Franco Calabrese and Donatella Zacco, who believed in me and guided me with my decisions. I would never be the way I am without them, in the good and in the bad things. Together with them, I would like to thank my dog Sheila. Although she will never read this work, she knows how much she means for me.

With these words, I conclude my academic career. So long, and thanks for all the fish.

References

- [1] The C++ Committee. The C++ standard. <https://isocpp.org/std/the-standard>, Last Access 8 January 2024.
- [2] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, USA, 2018.
- [3] Python Committee. Python official website. <https://www.python.org>, Last Access 8 January 2024.
- [4] Pierre Carbonnelle. Pypl official website. <https://pypl.github.io/PYPL.html>, Last Access 8 January 2024.
- [5] Python Software Foundation. Python documentation glossary. <https://docs.python.org/3/glossary.html>, Accessed December 2023.
- [6] Rob Pike. Concurrency is not parallelism. <https://go.dev/talks/2012/waza.slide#1>, 2021.
- [7] David P. Rodgers. Improvements in multiprocessor system design. *SIGARCH Comput. Archit. News*, 13(3):225–231, jun 1985.
- [8] Shavit Nir Herlihy Maurice. *The Art of Multiprocessor Programming*. Elsevier, 2012.
- [9] Edsger Dijkstra. Over seinpalen. Technical report, University of Texas at Austin, 1962 or 1963.
- [10] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. 22(3), 2003.
- [11] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8):391–407, 2012. APPLICATION ACCELERATORS IN HPC.
- [12] Vulkan. <https://www.vulkan.org>, 2024. Online; accessed 21 February 2024.
- [13] Vulkan kompute. <https://kompute.cc>, 2024. Online; accessed 21 February 2024.

- [14] Nvidia. CUDA C++ programming guide. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>, Last Access 8 January 2024.
- [15] Andrea Calabrese, Paolo Bernardi, Stefano Littardi, and Stefano Quer. Accelerated analysis of simulation dumps through parallelization on multicore architectures. In *Proceeding of the International Test Conference 2020*, 2020.
- [16] D. Appello, P. Bernardi, A. Calabrese, S. Littardi, G. Pollaccia, S. Quer, V. Tancorre, and R. Ugioli. Accelerated analysis of simulation dumps through parallelization on multicore architectures. In *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 69–74, 2021.
- [17] Andrea Calabrese. Tools for the analysis of simulation dumps and the evaluation of burn-in techniques. In *IEEE European Test Symposium (ETS) - 27th IEEE European Test Symposium (ETS)*, 2022.
- [18] D. Appello, Paolo Bernardi, Andrea Calabrese, G. Pollaccia, Stefano Quer, V. Tancorre, and R. Ugioli. Parallel multithread analysis of extremely large simulation traces. *IEEE Access*, 10:56440–56457, 2022.
- [19] F. Angione, P. Bernardi, A. Calabrese, L. Cardone, A. Niccoletti, D. Piumatti, S. Quer, D. Appello, V. Tancorre, and R. Ugioli. An innovative strategy to quickly grade functional test programs. In *2022 IEEE International Test Conference (ITC)*, pages 355–364, 2022.
- [20] Francesco Angione, Davide Appello, Paolo Bernardi, Andrea Calabrese, Stefano Quer, Matteo Sonza Reorda, Vincenzo Tancorre, and Roberto Ugioli. A toolchain to quantify burn-in stress effectiveness on large automotive system-on-chips. *IEEE Access*, 11:105655–105676, 2023.
- [21] Hideo Fujiwara and Shunichi Toida. The Complexity of Fault Detection Problems for Combinational Logic Circuits. *IEEE Transactions on Computers*, C-31:555–560, 1982.
- [22] Iso 26262-[1-10], road vehicles – functional safety. 2011.
- [23] Iliia Polian, Jens Anders, Steffen Becker, Paolo Bernardi, Krishnendu Chakrabarty, Nourhan ElHamawy, Matthias Sauer, Adit Singh, Matteo Sonza Reorda, and Stefan Wagner. Exploring the mysteries of system-level test. In *2020 IEEE 29th Asian Test Symposium (ATS)*, pages 1–6, 2020.
- [24] Harry H. Chen. Beyond structural test, the rising need for system-level test. In *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2018.
- [25] P. Varma. System chip test: are we there yet? In *Proceedings International Test Conference*, 1998.

- [26] T.M. Mak. Infant mortality—the lesser known reliability issue. In *13th IEEE International On-Line Testing Symposium (IOLTS 2007)*, pages 122–122, 2007.
- [27] M.F. Zakaria, Z.A. Kassim, M.P.-L. Ooi, and S. Demidenko. Reducing burn-in time through high-voltage stress test and weibull statistical analysis. *IEEE Design Test of Computers*, 23(2):88–98, 2006.
- [28] Alfredo Benso, Alberto Bosio, Stefano Di Carlo, Giorgio Di Natale, and Paolo Prinetto. Atpg for dynamic burn-in test in full-scan circuits. In *2006 15th Asian Test Symposium*, pages 75–82, 2006.
- [29] Davide Appello, Conrad Bugeja, Giorgio Pollaccia, Paolo Bernardi, Riccardo Cantoro, Marco Restifo, Ernesto Sanchez, and Federico Venini. An optimized test during burn-in for automotive soc. *IEEE Design Test*, 35(3):46–53, 2018.
- [30] F. Almeida et al. Effective screening of automotive socs by combining burn-in and system level test. In *IEEE International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2019.
- [31] F. Angione, P. Bernardi, G. Filippini, M. Sonza Reorda, D. Appello, V. Tancorre, and R. Ugioli. An optimized burn-in stress flow targeting interconnections logic to embedded memories in automotive systems-on-chip. In *2022 IEEE European Test Symposium (ETS)*, pages 1–6, May 2022.
- [32] Paolo Bernardi, Alberto Bosio, Giorgio Di Natale, Andrea Guerriero, Ernesto Sanchez, and Federico Venini. Improving Stress Quality for SoC Using Faster-than-At-Speed Execution of Functional Programs. In Thomas Hollstein, Jaan Raik, Sergei Kostin, Anton Tšertov, Ian O’Connor, and Ricardo Reis, editors, *VLSI-SoC: System-on-Chip in the Nanoscale Era – Design, Verification and Reliability*, volume AICT-508 of *IFIP Advances in Information and Communication Technology*, pages 130–151, Tallinn, Estonia, September 2016. Springer International Publishing.
- [33] Chen He and Yanyao Yu. Wafer level stress: Enabling zero defect quality for automotive microcontrollers without package burn-in. In *2020 IEEE International Test Conference (ITC)*, pages 1–10, Nov 2020.
- [34] Sun-Jung Lee, Soo-Geun Lee, Bong-Suk Suh, Hongjae Shin, Nae-In Lee, Ho-Kyu Kang, and Gwangpyuk Suh. New insight into stress induced voiding mechanism in cu interconnects. In *Proceedings of the IEEE 2005 International Interconnect Technology Conference, 2005.*, pages 108–110, June 2005.
- [35] Walter Ruggeri, Paolo Bernardi, Stefano Littardi, Matteo Sonza Reorda, Davide Appello, Claudia Bertani, Giorgio Pollaccia, Vincenzo Tancorre, and Roberto Ugioli. Innovative methods for burn-in related stress metrics computation. In *2021 16th International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6, 2021.

- [36] D. Appello, P. Bernardi, R. Cagliesi, M. Giancarlini, M. Grosso, E. Sanchez, and M. Sonza Reorda. Automatic functional stress pattern generation for soc reliability characterization. In *2009 14th IEEE European Test Symposium*, pages 93–98, May 2009.
- [37] Dietmar Vogel, Sven Rzepka, Bernd Michel, and Astrid Gollhardt. Local stress measurement on metal lines and dielectrics of beol pattern by stress relief technique. In *2011 Semiconductor Conference Dresden*, pages 1–3, 2011.
- [38] Rabindra K. Roy, T.M. Niermann, Janak H. Patel, Jacob A. Abraham, and Resve A. Saleh. Compaction of ATPG-generated Test Sequences for Sequential Circuits. *IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 382–385, 1988.
- [39] Che-Jen Jerry Chang and Takeo Kobayashi. Test Quality Improvement with Timing-aware ATPG: Screening small delay defect case study. *IEEE International Test Conference*, pages 1–1, 2008.
- [40] Sounil Biswas and Bruce Cory. An Industrial Study of System-Level Test. *IEEE Design Test of Computers*, 29(1):19–27, 2012.
- [41] IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–590, 2006.
- [42] Stuart Sutherland. *Extended VCD files*, pages 104–105. Springer US, Boston, MA, 2002.
- [43] Cadence. Cadence website. https://www.cadence.com/en_US/home.html, Accessed March 2024.
- [44] GTKWave development team. Gtkwave. <https://github.com/gtkwave/gtkwave>, Accessed March 2024.
- [45] Richard Hipp. Sqlite. <https://sqlite.org>, Accessed March 2024.
- [46] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *12th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, volume 32 of *ACM SIGPLAN Notices*, pages 125–141, Atlanta, United States, October 1997. ACM SIGPLAN, ACM Press. Cette publi est considérée comme un journal (ACM SIGPLAN Notices) dans la communauté.
- [47] Stephen C. Johnson. Yacc. <https://www.tuhs.org/cgi-bin/utree.pl?file=V6/usr/source/yacc>.
- [48] Free Software Foundation. Gnu bison. <https://www.gnu.org/software/bison/>.

- [49] STMicroelectronics. Spc58nn84c3, 32-bit power architecture mcu for high performance applications. <https://www.st.com/en/automotive-microcontrollers/spc58nn84c3.html>.
- [50] Michael Hahsler, Matthew Piekenbrock, and Derek Doran. dbSCAN: Fast density-based clustering with R. *Journal of Statistical Software*, 91(1):1–30, 2019.
- [51] Sdl libraries. <https://www.libsdl.org>, 2024. Online; accessed 21 February 2024.
- [52] Openmp. <https://www.openmp.org>, 2024. Online; accessed 21 February 2022.
- [53] Francesco Angione et al. Test, Reliability and Functional Safety trends for Automotive System-on-Chip. European Test Symposium, 2022.
- [54] F. Angione, D. Appello, P. Bernardi, C. Bertani, G. Gallo, S. Littardi, G. Pollaccia, W. Ruggeri, M. Sonza Reorda, V. Tancorre, and R. Ugioli. A low-cost burn-in tester architecture to supply effective electrical stress. *IEEE Transactions on Computers*, pages 1–14, 2022.
- [55] C++ named requirements: Trivially copyable. https://en.cppreference.com/w/cpp/named_req/TriviallyCopyable, 2024. Online; accessed 12 February 2024.
- [56] Muhammad Hassan et al. Early soc security validation by vp-based static information flow analysis. In *IEEE/ACM ICCAD*, 2017.
- [57] Rolf Drechlsler et al. Ensuring correctness of next generation devices: From reconfigurable to self-learning systems. In *IEEE ATS*, 2019.
- [58] Wei Hu et al. An overview of hardware security and trust: Threats, countermeasures, and design tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.
- [59] Khitam M Alatoun et al. Efficient methods for soc trust validation using information flow verification. In *IEEE ICCD*, 2021.
- [60] Paolo Bernardi et al. Development flow for on-line core self-test of automotive microcontrollers. *IEEE Transactions on Computers*, 2016.
- [61] D. Piumatti et al. An efficient strategy for the development of software test libraries for an automotive microcontroller family. *Microelectronics Reliability vol. 115*, 2020.
- [62] Sharad Malik et al. Specification and modeling for systems-on-chip security verification. In *Proceedings of Design Automation Conference*, 2016.
- [63] F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero. Efficient machine-code test-program induction. In *CEC*, 2002.

- [64] Jean J. Labrosse. *UC/OS-III, The Real-Time Kernel, or a High Performance, Scalable, ROMable, Preemptive, Multitasking Kernel for Microprocessors, Microcontrollers & DSPs*. Micrium Press, 2009.
- [65] Alessandro Borione, Lorenzo Cardone, Andrea Calabrese, and Stefano Quer. An experimental evaluation of graph coloring heuristics on multi- and many-core architectures. *IEEE Access*, 11:125226–125243, 2023.
- [66] SCITEPRESS, editor. *A Web Scraping Algorithm to Improve the Computation of the Maximum Common Subgraph*, 2024.
- [67] Frank Thomson Leighton. A graph coloring algorithm for large scheduling problems. *Journal of research of the national bureau of standards*, 84(6):489, 1979.
- [68] Dominic JA Welsh and Martin B Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–86, 1967.
- [69] Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981.
- [70] Fusun Akman. Partial chromatic polynomials and diagonally distinct sudoku squares. *arXiv preprint arXiv:0804.0284*, 2008.
- [71] Krzysztof Giaro, Marek Kubale, and Pawel Obszarski. A graph coloring approach to scheduling of multiprocessor tasks on dedicated machines with availability constraints. *Discrete Applied Mathematics*, 157(17):3625–3630, 2009.
- [72] M. Garey and D. Johnson. *Computers and Intractability – A Guide to the Theory of NP-completeness*. Freeman, 1979.
- [73] Assefaw Hadish Gebremedhin and Fredrik Manne. Scalable parallel graph coloring algorithms. *Concurrency: Practice and Experience*, 12(12):1131–1146, 2000.
- [74] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.
- [75] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669, 1993.
- [76] Jonathan Cohen and Patrice Castonguay. Efficient graph matching and coloring on the gpu. In *GPU Technology Conference*, pages 1–10, 2012.
- [77] Alessandro Garbo and Stefano Quer. A Fast MPEG’s CDVS Implementation for GPU Featured in Mobile. *IEEE Access*, 6(1):52027–52046, dec 2018.

- [78] Stefano Quer, Marcelli Andrea, and Squillero Giovanni. The Maximum Common Subgraph Problem: A Parallel and Multi-Engine Approach. *MDPI Computation*, 8(2):1–29, 2020.
- [79] Muhammad Osama, Minh Truong, Carl Yang, Aydın Buluç, and John Owens. Graph coloring on the gpu. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 231–240. IEEE, 2019.
- [80] Yuxin Chen, Benjamin Brock, Serban Porumbescu, Aydın Buluc, Katherine Yelick, and John Owens. Atos: A task-parallel gpu scheduler for graph analytics. In *Proceedings of the 51st International Conference on Parallel Processing, ICPP '22*, New York, NY, USA, 2023. Association for Computing Machinery.
- [81] NVIDIA Corporation. cusparse library documentation. <https://docs.nvidia.com/cuda/cusparse/index.html>.
- [82] Muhammad Osama, Serban D. Porumbescu, and John D. Owens. Essentials of Parallel Graph Analytics. In *Proceedings of the Workshop on Graphs, Architectures, Programming, and Learning, GrAPL 2022*, pages 314–317, 5 2022.
- [83] Jonathan Cohen. Proof of optimality of minmax pis algorithm. 2011.
- [84] Muhammad Osama. Private Communication, jun 2022.
- [85] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011.
- [86] Carla Gomes. <https://mat.gsia.cmu.edu/COLOR02/>, apr 2022.
- [87] Benedek Rozemberczki and Rik Sarkar. Twitch gamers: a dataset for evaluating proximity preserving and structural role-based node embeddings, 2021.
- [88] Andrew Dalke and Janna Hastings. Fmcs: a novel algorithm for the multiple mcs problem. *Journal of cheminformatics*, 5(Suppl 1):O6, 2013.
- [89] Stanley Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.
- [90] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1):107–117, 1998. Proceedings of the Seventh International World Wide Web Conference.
- [91] Younghee Park and Douglas Reeves. Deriving common malware behavior through graph clustering. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 497–502, 2011.

- [92] Thomas Zimmermann and Nachiappan Nagappan. Predicting subsystem failures using dependency graph complexities. In *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*, pages 227–236, 2007.
- [93] F. Angione, P. Bernardi, A. Calabrese, L. Cardone, A. Niccoletti, D. Piumatti, S. Quer, D. Appello, V. Tancorre, and R. Ugioli. An innovative strategy to quickly grade functional test programs. In *2022 IEEE International Test Conference (ITC)*, pages 355–364, 2022.
- [94] Coenraad Bron and Joep Kerbosch. Finding All Cliques of an Undirected Graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.
- [95] Harry G. Barrow and Rod M. Burstall. Subgraph Isomorphism, Matching Relational Structures and Maximal Cliques. *Inf. Process. Lett.*, 4(4):83–84, 1976.
- [96] Ciaran McCreesh, Patrick Prosser, and James Trimble. A partitioning algorithm for maximum common subgraph problems. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 712–719, 2017.
- [97] Yanli Liu, Chu-Min Li, Hua Jiang, and Kun He. A learning based branch and bound for maximum common subgraph related problems. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(03):2392–2399, Apr. 2020.
- [98] Jianrong Zhou, Kun He, Jiongzhi Zheng, Chu-Min Li, and Yanli Liu. A strengthened branch and bound algorithm for the maximum common (connected) subgraph problem, 2022.
- [99] Yanli Liu, Jiming Zhao, Chu-Min Li, Hua Jiang, and Kun He. Hybrid learning with new value function for the maximum common subgraph problem, 2022.
- [100] P. Foggia, C. Sansone, and M. Vento. A database of graphs for isomorphism and sub-graph isomorphism benchmarking. In -, page 176–187, 2001.
- [101] Uwe Schöning. Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences*, 37(3):312–323, 1988.
- [102] David S. Johnson Michael Garey. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, United States, 1979.
- [103] James Trimble. *Partitioning algorithms for induced subgraph problems*. PhD thesis, University of Glasgow, 2023.
- [104] G. Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *CALCOLO*, 9(4):341–352, 1973.

- [105] Ciaran McCreesh, Samba Ndojh Ndiaye, Patrick Prosser, and Christine Solnon. Clique and constraint models for maximum common (connected) subgraph problems. In Michel Rueher, editor, *Principles and Practice of Constraint Programming*, pages 350–368, Cham, 2016. Springer International Publishing.
- [106] Philippe Vismara and Benoît Valery. Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms. In *Modelling, Computation and Optimization in Information Systems and Management Sciences: Second International Conference MCO 2008, Metz, France-Luxembourg, September 8-10, 2008. Proceedings*, pages 358–368. Springer, 2008.
- [107] Stefano Quer, Andrea Marcelli, and Giovanni Squillero. The maximum common subgraph problem: A parallel and multi-engine approach. *Computation*, 8(2), 2020.
- [108] Lorenzo Cardone and Stefano Quer. The multi-maximum and quasi-maximum common subgraph problem. *Computation*, 11(4), 2023.
- [109] Rafael Martí and Gerhard Reinelt. *Heuristic Methods*, pages 27–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2022.
- [110] Squillero Calabrese, Quer and Tonda. Towards an evolutionary approach for exploiting core knowledge in artificial intelligence. Accepted in GECCO 2024, 2024.
- [111] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- [112] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, May 2017.
- [113] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstern, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, July 2021.
- [114] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John

- Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.
- [115] Gary Marcus. The next decade in AI: Four steps towards robust artificial intelligence, 2020.
- [116] Waddah Saeed and Christian Omlin. Explainable AI (XAI): A systematic meta-survey of current challenges and future opportunities. *Knowledge-Based Systems*, 263:110273, March 2023.
- [117] Pietro Barbiero, Gabriele Ciravegna, Francesco Giannini, Mateo Espinosa Zarlenga, Lucie Charlotte Magister, Alberto Tonda, Pietro Lio, Frederic Precioso, Mateja Jamnik, and Giuseppe Marra. Interpretable neural-symbolic concept reasoning. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 1801–1825. PMLR, 23–29 Jul 2023.
- [118] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps, 2014.
- [119] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. “why should i trust you?”: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’16. ACM, August 2016.
- [120] Tarek R. Besold, Artur d’Avila Garcez, Sebastian Bader, Howard Bowman, Pedro Domingos, Pascal Hitzler, Kai-Uwe Kühnberger, Luis C. Lamb, Priscila Machado Vieira Lima, Leo de Penning, Gadi Pinkas, Hoifung Poon, and Gerson Zaverucha. Neural-symbolic learning and reasoning: A survey and interpretation1. In *Frontiers in Artificial Intelligence and Applications*. IOS Press, December 2021.
- [121] François Chollet. On the measure of intelligence, 2019.
- [122] Elizabeth S. Spelke and Katherine D. Kinzler. Core knowledge. *Developmental Science*, 10(1):89–96, January 2007.
- [123] Noam Chomsky et al. On cognitive structures and their development: A reply to piaget. *Language and learning: the debate between Jean Piaget and Noam Chomsky*, pages 35–54, 1980.
- [124] Pierre Pica, Cathy Lemer, Véronique Izard, and Stanislas Dehaene. Exact and approximate arithmetic in an amazonian indigene group. *Science*, 306(5695):499–503, October 2004.

- [125] Stanislas Dehaene, Véronique Izard, Elizabeth Spelke, and Pierre Pica. Log or linear? distinct intuitions of the number scale in western and amazonian indigene cultures. *Science*, 320(5880):1217–1220, May 2008.
- [126] Fei Xu and Elizabeth S. Spelke. Large number discrimination in 6-month-old infants. *Cognition*, 74(1):B1–B11, January 2000.
- [127] Steven Pinker and Jacques Mehler, editors. *Connections and Symbols*. Connections and Symbols. MIT Press, London, England, January 1988.
- [128] E Spelke, N Kanwisher, and J Duncan. *Functional Neuroimaging of Visual Cognition: Attention and Performance*, volume 20. Oxford University Press, 2004.
- [129] George Boole. *An investigation of the laws of thought on which are founded the mathematical theories of logic and probabilities by george boole*. Walton and Maberly, 1854.
- [130] Melanie Mitchell. Abstraction and analogy-making in artificial intelligence. *Annals of the New York Academy of Sciences*, 1505(1):79–101, June 2021.
- [131] John Holland. *Progress in theoretical biology*, chapter Adaptation. New York: Academic Press, 1976.
- [132] Stewart W Wilson. Classifier fitness based on accuracy. *Evolutionary computation*, 3(2):149–175, 1995.
- [133] Martin V Butz. *Rule-based evolutionary online learning systems*, volume 259. Springer, 2006.
- [134] Iztok Fister Jr. and Iztok Fister. *A Brief Overview of Swarm Intelligence-Based Algorithms for Numerical Association Rule Mining*, pages 47–59. Springer Singapore, Singapore, 2021.
- [135] Sreejan Kumar, Carlos G. Correa, Ishita Dasgupta, Raja Marjeh, Michael Y. Hu, Robert D. Hawkins, Nathaniel D. Daw, Jonathan D. Cohen, Karthik Narasimhan, and Thomas L. Griffiths. Using natural language and program abstractions to instill human inductive biases in machines. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35 - 36th Conference on Neural Information Processing Systems, NeurIPS 2022*, Advances in Neural Information Processing Systems. Neural information processing systems foundation, 2022.
- [136] Felix Haase and Diedrich Wolter. Behind the corner: Using qualitative reasoning for solving angry birds.
- [137] Xiaoyu Ge, Jae Lee, Jochen Renz, and Peng Zhang. Hole in one: Using qualitative reasoning for solving hard physical puzzle problems,. 06 2016.

-
- [138] A. M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.
- [139] Charles B. Craighead, W. Edward; Nemeroff. *The Concise Corsini Encyclopedia of Psychology and Behavioral Science*. John Wiley & Sons, 2004.
- [140] Kendall Haven and Donna Clark. *100 Most Popular Scientists for Young Adults: Biographical Sketches and Professional Paths*. Libraries Unlimited, May 1999.
- [141] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.
- [142] Alberto Tonda. Inspyred: Bio-inspired algorithms in python. *Genetic Programming and Evolvable Machines*, 21(1):269–272, 2020.
- [143] Gary J. Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (hevc) standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649–1668, Dec 2012.
- [144] Joel Lehman and Kenneth O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19:189–223, 6 2011. Fundamental reference for Novelty Search.